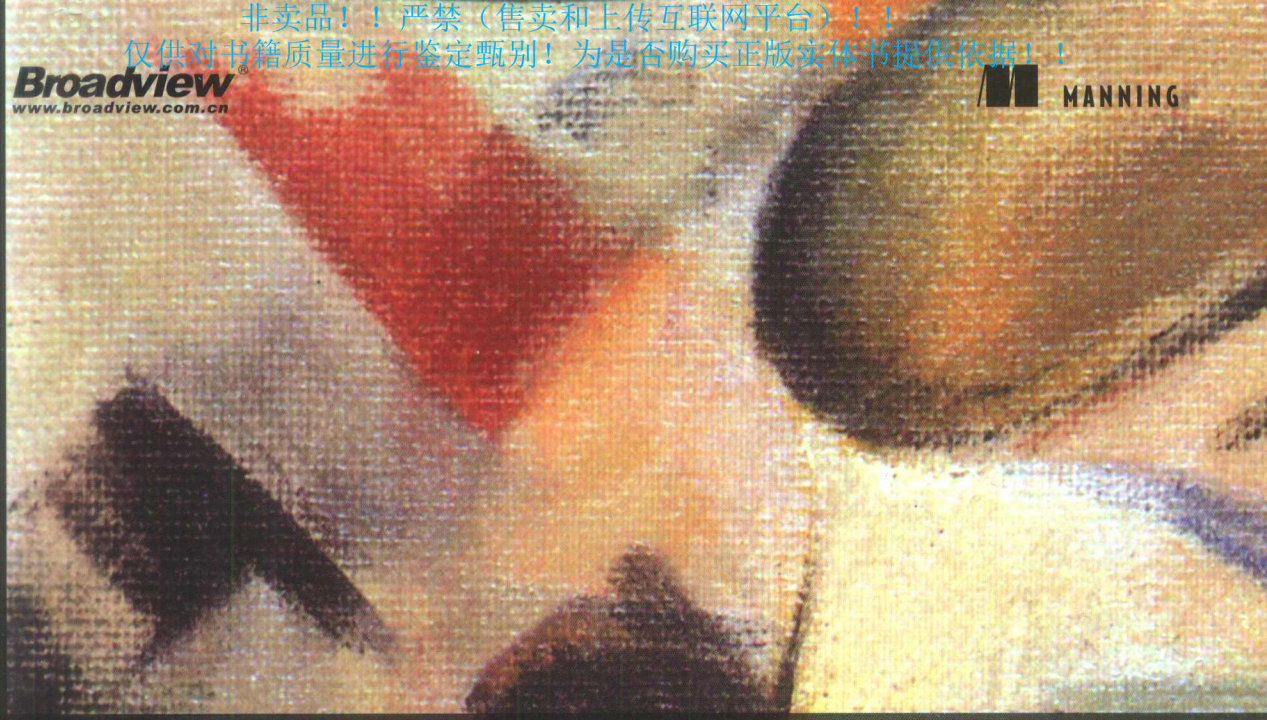
Broadview
www.broadview.com.cn

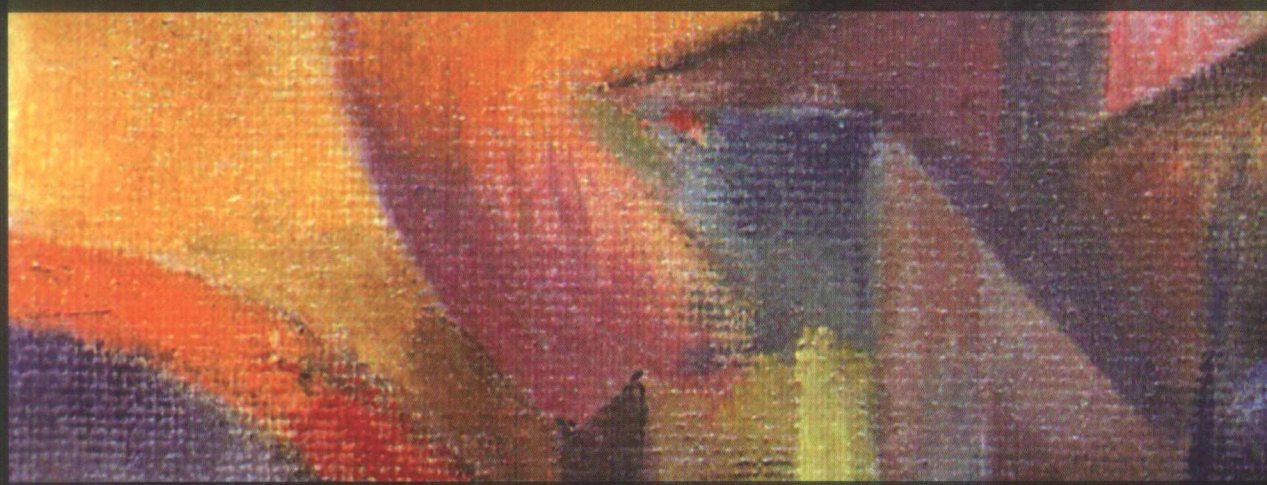
MANNING



Functional and Reactive Domain Modeling

函数响应式领域建模

[美] Debasish Ghosh 著
李 源 译



中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Functional and Reactive Domain Modeling

函数响应式领域建模

[美] Debasish Ghosh 著
李 源 译

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

传统的分布式应用不会切入微服务、快速数据及传感器网络的响应式世界。为了捕获这些应用的动态联系及依赖，我们需要使用另外一种方式来进行领域建模。由纯函数构成的领域模型是以一种更加自然的方式来反映一个响应式系统内的处理流程，同时它也直接映射到了相应的技术和模式，比如 Akka、CQRS 以及事件溯源。本书讲述了响应式系统中建立领域模型所需要的通用且可重用的技巧——首先介绍了函数式编程和响应式架构的相关概念，然后逐步地在领域建模中引入这些新的方法，同时本书提供了大量的案例，当在项目中应用这些概念时，可作为参考。

版权贸易合同登记号 图字：01-2016-9180

图书在版编目（CIP）数据

函数响应式领域建模 /（美）德巴斯什 • 戈施（Debasish Ghosh）著；李源译. —北京：电子工业出版社，2018.1
书名原文：Functional and Reactive Domain Modeling
ISBN 978-7-121-32392-8

Ⅰ. ①函… Ⅱ. ①德… ②李… Ⅲ. ①函数－程序设计 Ⅳ. ①TP311.1

中国版本图书馆CIP数据核字（2017）第185481号

责任编辑：张春雨
印　　刷：三河市双峰印刷装订有限公司
装　　订：三河市双峰印刷装订有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编：100036
开　　本：787×980　1/16　印张：18.5　字数：383.6 千字
版　　次：2018 年 1 月第 1 版
印　　次：2018 年 1 月第 1 次印刷
定　　价：79.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

推荐序

开发人员正淹没在各种错综复杂的问题中，需要借助多核处理器以及分布式基础架构的优势，来应对产生数据越来越多的高要求用户规模的迅猛增长，以确保更低的延迟以及更高的吞吐率。所以开发人员不得不在消费者日益苛刻的紧张截止时间前按时交付。

开发人员的工作从来没有轻松过。为了能保持多产的同时又能享受工作，需要采用合适的工具集——这些工具可以通过优化资源的使用来管理日益增长的复杂性以及需求。通常，并不是简单地追逐最新、最炫的东西——尽管这很诱人。所以必须要回顾总结，从过去艰难获胜的经验中学习，看是否可以将其应用到今天的场景以及挑战中。我认为开发人员开发的那些非常有用的工具中所包含的领域驱动设计（domain-driven design，DDD）、函数式编程（FP）以及响应式原则，都可以帮助我们管理复杂事务的某个方面。

- 领域复杂性：领域驱动设计帮助我们挖掘并理解领域的不同特性与语义。通过跟利益相关方用他们的语言进行沟通，DDD 可以更容易地创建可扩展的领域模型来映射真实世界，同时允许持续的变化。
- 解决方案复杂性：函数式编程可以帮助我们保持合理性及可组合性。通过可重用的纯函数并使用稳定（不可变）值，函数式编程提供了一个伟大的工具集，通过不会“撒谎”的代码来得出运行时间、并发性以及抽象过程。

- 系统复杂性：正如在 *The Reactive Manifesto*（http://www.reactivemanifesto.org）中所定义的，响应式原则能帮助我们管理日益复杂的世界，包括多核处理器、云计算、移动设备以及物联网。在这里，所有新系统本质上都是分布式系统。要运作这个世界是非常困难而且很有挑战的，但同样，也拥有很多有趣的新的机会。这种变化迫使我们的行业去反思过去一些围绕系统架构以及设计方面的最佳实践。

我非常喜欢阅读这本书，它完全体现了我在过去十几年的自身经历。我从 OO 实习生开始——白天埋头于 C++ 和 Java，而晚上阅读经典的 *Gang of Four*[1]。2006 年我开始阅读 Eric Evan 关于领域驱动建模的书[2]，它对我或多或少有所启发。然后我就变成一个 DDD 狂热爱好者，在所有可能的地方去应用它。多年后，我又开始使用 Erlang[3]，然后是 Scala[4]，它们都让我再次感受到了函数式编程的魅力并深深地爱上了它。我在大学期间学过函数式编程，但当时并没有真正意识到它的威力。在这段时间里，我开始对 Java 在并发性、适应性以及可伸缩性方面的“最佳实践”逐渐失去信仰。在 Erlang 方式，特别是 actor 模型[5]（我认为这是一个更好的做事方式）的指引下，我开始了 Akka 项目，我相信这会有助于将响应式原则带入主流。

这本书之所以能吸引我是因为它设立了一个更加宏大的目标，将 3 个完全不同的工具（领域驱动设计、函数式编程以及响应式原则）用可实践的方式整合到了一起。它教会你诸如边界上下文、领域事件、函数、monad、applicative、future、actor、流以及 CQRS[6] 等内容是如何使复杂性保持可控的。如果内心不够强大，那么这本书将不适合你，阅读它很费劲。但如果花上数小时，你就会收获一些基础概念。亲爱的读者，幸运的你已经迈出了第一步，接下来所需要做的就是继续读下去。

JONAS BONÉR
Lightbend 创始人兼 CFO
Akka 创始人

---

1 Erich Gamma、Richard Helm、Ralph Johnson与John Vlissides所著的*Design Patterns: Elements of Reusable Object-Oriented Software*（Addison-Wesley, 1994年），也是我们常说的*Gang of Four*。

2 Eric Evans所著的*Domain-Driven Design: Tackling Complexity in the Heart of Software*（Addison-Wesley, 2003年）。

3 一种面向并发的编程语言，由瑞典电信设备制造商爱立信所辖的CS-Lab开发。——译者注

4 一种类似Java的多范式编程语言，设计初衷是实现可伸缩的语言，并集成面向对象编程和函数式编程的各种特性。——译者注

5 https://en.wikipedia.org/wiki/Actor_model。

6 command query responsibility segregation，命令查询责任分离。——译者注

序

在2014年夏天，Manning出版社希望出版*DSLs in Action*（https://www.manning.com/books/dsls-in-action）的升级版本，因为DSL的所有新特性都围绕编程语言的设计和实现。巧合的是正好在那个时间，我用函数模式对一个复杂的领域模型成功地进行了重构。

跟一群刚毕业进入Scala函数式编程世界的软件工程师们一起，我将域行为建模为纯粹的函数，将域对象设计为代数数据类型，并开始意识到代数API设计的价值。团队的每个成员人手一本Paul Chiusano和Rúnar Bjarnason刚完成的*Functional Programming in Scala*（中文版为《Scalo函数式编程》，由电子出版社出版）。

我们的域模型非常复杂，实现严格遵守Eric Evans在他的著作*Domain-Driven Design: Tackling Complexity in the Heart of Software*（Addison-Wesley，2003年）中所阐述的领域驱动设计（DDD）的原则。不过我们没有用面向对象的方式，而是决定采用函数式编程。一切的开始都像是一个实验，但在最后证明这是一次非常成功并且令人满意的经历。现在当我回头看时，发现DDD的内容与软件工程的通用规则非常协调一致。因此也不用担心函数式、领域驱动设计会显得像是领域建模的典型范例。

这本书是我们成功运用函数式编程进行领域模型开发的证据。我决定跟读者分享我们遵守的实践、采用的原则，以及在实现中所使用的Scala风格。Manning出

版社完全同意这个想法并决定继续该项目。

不管你的领域模型是什么样的，定义实现成功的一个关键标准是应用的响应能力。没有一个用户喜欢盯着屏幕上的等待光标，根据我们的经验来看，这通常是因为架构师非必要地阻塞了主线程的执行。需要花费时间执行的昂贵的操作应该用异步的方式来执行，把主线程空出来给其他用户行为。*The Reactive Manifesto*（www.reactivemanifesto.org）中定义了建模所需要使用的特性，以便保证应用程序是非阻塞、响应及时的，并避免巨大延迟带来的恶劣影响。这也是我要在书中写的另一个方面。在经过与 Manning 团队多次友好的商讨后，我们决定在这本书中将函数与响应式编程结合起来。

于是本书就诞生了。通过这个项目，我收获了巨大的乐趣，也希望读者能有类似的体验。我收到了无数读者、评审者、良好祝愿者们的留言，他们陪着我一起提升了这本书的质量。我也非常感谢来自 Manning 出版社经验丰富的编辑以及评审者团队的巨大支持。

致谢

我要感谢很多人，他们直接或间接地参与了这本书的创作。

首先，我要感谢 Martin Odersky，Scala 编程语言的创建者，我用 Scala 完成了所有函数响应式领域建模的案例。同时也非常感谢你建立了 Scalaz，这个有趣的库使我们在用 Scala 语言进行纯函数编程时充满乐趣。

Twitter 是一个非常酷的沟通方式，承载了各种各样的讨论。我在上面和一些牛人就函数式编程有过很多非常激烈的讨论。感谢每一位牛人，是你们促使我完成了这本书。

感谢所有的评审者：Barry Alexander、Cosimo Attanasi、Daniel Garcia、Jan Nonnen、Jason Goodwin、Jaume Valls、Jean-François Morin、John G. Schwitz、Ken Fricklas、Lukasz Kupka、Michael Hamrah、Othman Doghri、Rintcius Blok、Robert Miller、Saleem Shafi、Tarek Nabil，以及 William E. Wheeler。时间可能是我们拥有的最宝贵的资源，我非常感谢他们愿意在这本书上花费时间，每个评审者都给了我很棒的建议，极大地提升了这本书的质量。

感谢所有购买了 MEAP[1] 的读者，在作者在线论坛里的定期沟通，一直鼓励着我完成这本书。特别要感谢 Arya Irani，她贡献的一个 pull 请求帮助我更新了 monad

1 Manning Early Access Program，在书的写作过程中，你可以阅读到刚完成的章节，并第一时间得到最终版本的电子版。——译者注

代码（从基于 Scalaz 7.1 到 7.2）。同样要特别感谢 Thomas Lockney 和 Charles Feduke，他们对每个不同的 MEAP 版本做了彻底的技术评审。

我还要感谢 Manning 出版社再次信任我。在我写第一本书的时候，我们有过非常美好的合作，而再次合作甚至更有乐趣。我要感谢以下 Manning 员工的杰出工作。

- 感谢 Michael Stephens 和 Christina Rudloff 促使我启动这个项目。
- 感谢 Jennifer Stout 在 10 个章节的漫长过程中不屈不挠地纠正了我所有的错误。
- 感谢 Alain Gouniot 在整个过程中提供了深入的技术评审。
- 感谢 Gandace Gilhooley 与 Ana Romac 帮助推动这本书。
- 感谢 Mary Piergies、Kevin Sullivan、Maureen Spencer，以及所有幕后工作人员（包括 Sharon Wilkey、Alyson Brener、April Milne，以及 Dennis Dalinnik），他们帮助我把一个粗糙的草稿变成一本真正的书。

感谢 Jonas Bonér 为我的书写序。我很荣幸，我与 Jonas 已经相识了很长时间，他也是我很多软件开发项目的重要灵感来源。

最后，我要感谢我的妻子、母亲以及我的儿子 Aarush，他们给我提供了最完美的“生态环境”，在那里，写一本关于函数式编程的书这种创造性任务才有可能完成。

关于本书

本书内容涉及如何使用函数式编程实现领域模型，以及如何通过使用响应式原则（诸如非阻塞计算和异步消息）来确保模型的响应性。

领域模型都是针对问题领域的，可以通过很多方式实现一个解决方案框架——能提供与问题领域模型相同的函数性，通常会使用面向对象技术来设计领域模型。本书中使用了一种正交方式——用纯函数对领域行为建模，用代数数据类型对领域实体建模，并将不变性作为设计空间的一个主关注点。作为读者，你能学到基于代数技术的函数式设计模式，可以将其直接用于实现自己的领域模型。

这本书同样还包括了响应式编程——使用 future、promise、actor 以及 stream 来确保模型在有一定延迟的条件下有足够的响应性和可操作性。

书中使用 Scala 语言来实现领域模型。作为 JVM 的一个“常驻民”，在面向对象以及函数式编程准则的强力支持下，今天 Scala 已经是最广泛使用的语言之一。尽管如此，本书讨论的核心准则同样适用于其他函数式语言，比如 Haskell。

内容简介

第 1 章会对从书中能学习到什么做一个全面的讨论。这一章中会提供一个关于领域模型的概览，同时讨论一些领域驱动设计背后的概念。也会谈到函数式编程（FP）的核心原则，以及将领域模型设计得足够透明并且根据纯逻辑进行边界解耦后所能

得到的好处。这一章还定义了*响应式模式*，包括如何将 FP 和响应式设计这两个概念捆绑在一起使得模型更具有响应性和可伸缩性。

第 2 章会讨论用 Scala 作为函数响应式领域建模实现语言的好处。它会讨论静态类型的好处，以及 Scala 的高级类型系统如何让模型更加健壮并经得起考验。在这一章中，还会学到如何将 OO 和 FP 的力量结合起来实现模块化的干净的模型。

第 3 章从讨论一个代数 API 设计开始。先不考虑实现，可以基于抽象的代数来设计 API。这一章结合大量的细节以及现实世界中个人银行系统建模的一些例子，呈现了这种方式的优点。代数有它自身的法则，基于代数建设 API 时，要确保实现遵守这些法则。这一章以一些关于领域对象生命周期的讨论作为结尾，从工厂中生产出来开始，然后执行领域行为，最后持久化。

第 4 章聚焦在函数式设计模式上，这跟以前所学的面向对象的设计模式有极大的不同。一个函数式设计模式是基于代数方法的，它可以有很多种实现方式（或诠释），因此在重用性上会远超过 OO 设计模式。这一章将讨论 functor、applicative、monad，这些都是函数式编程语言中的最基本的可复用模式。这一章还会讨论一些使用案例，如基于这些模式的代数方法如何演变领域模型。

第 5 章是关于如何模块化领域模型。一个非凡的领域模型是一系列小模型的集合，每个小模型都被认为是边界上下文（bounded context）。这一章会解释如何将边界上下文设计成独立的加工品，以及如何确保多个边界上下文之前的通信在空间和时间上的解耦。这是领域驱动设计的核心概念之一，而且可以用异步消息来很容易地实现。本章还将介绍 free monad，另一种运用函数式编程概念的高级模块化技术。

第 6 章将讨论响应式领域模型。这一章涉及如何设计响应式 API，既不会阻塞主线程的执行，同时又能保证模型的响应性。这一章会提供领域对象和边界上下文（如 future、promise、actor 和 reactive stream）之间非阻塞式通信的各种方式，还会讨论一个使用场景，即在个人银行领域中如何运用 reactive stream。

第 7 章讲解了 reactive stream。用 Akka Stream 实现了一个中等规模的用例来证明 reactive stream 的威力。第 6 章涉及了 actor 模型的缺点，而第 7 章则告诉我们如何用 Akka Stream 实现类型 API 来克服这些缺点。

第 8 章覆盖了领域模型的持久化（persistence）。这一章从一个基于 CRUD[1] 持久化模型的评论开始，介绍运用事件驱动技术的响应式持久化的概念。这一章结合了当前模型的状况介绍领域事件的完整历史，讨论诸如 CQRS 和事件源的实现技术，这也产生了一个更有弹性的持久化模型。这一章也用 Slick[2] 演示了一个基于 CRUD 的实现，一种针对 RDBMS 的通用的函数型到关系型映射框架。

---

1 Create、Retrieve、Update、Delete。——译者注

2 一种响应式的jQuery插件。——译者注

第 9 章是关于测试领域模型的。它从基于 xUnit[1] 经典的测试方法论开始，然后列出其中的缺陷以及如何使用代数测试进行改进。这一章介绍了基于属性的测试，它允许用户编写代数属性，然后通过运行时自动生成的数据来验证它。这一章会使用 Scala 基于属性的测试库 ScalaCheck，结合前面章节已有的领域模型来讨论这种技术的实现方式。

本书最后在第 10 章中对核心概念进行回顾，并讨论领域建模未来的发展趋势。

代码约定及下载

书中所有源代码都用等宽字体来和普通文本区分。有时需要将一行代码分成两行甚至更多行来适应书页。我们会用这样的箭头➥来表示连续行。

代码中包含很多注释。在一些情况下，会用数字标记与文后的注解对应关联。

书中案例的代码可以从出版社网站 https://www.manning.com/books/functional-and-reactive-domain-modeling 以及 GitHub 网址 https://github.com/debasishg/frdomain 上进行下载。

测验与练习

书中包含一系列测验与练习，它们会帮助读者了解自身对讨论内容的理解程度。第 1 章和第 2 章包含一些基本概念的测验。每个“测验时间”之后的一页或两页就会有相应的“测验答案”，如以下例子所示。

测验时间 1.1　你认为这种模型最主要的缺点是什么？

测验答案 1.1　主要问题是易变性，它会从两个方面打击你：很难在并行设置下使用抽象，而且很难推理你的代码。

从第 3 章开始，练习会复杂一些。测验被编号的练习所替代——实际的建模问题，这些练习将聚焦在对应章节所讨论的概念上。练习如同以下例子。

练习 3.2 验证透镜规律

第 3 章的在线代码库中包含一个 `Customer` 实体的定义以及它的透镜。观察 `addressLens`，它将更新一个用户的地址，然后用 ScalaCheck 写出属性来验证透镜规律。

针对解决方案领域的一个特定用例建模可以通过不同的途径来完成。练习会讨

1　如JUnit。——译者注

论这些候选方案以及每个方案的优缺点。我们鼓励读者独立完成它们而不是直接看解决方案，这些方案同样可以在出版社网站（https://www.manning.com/books/functional-and-reactive-domain- modeling）和 GitHub（https://github.com/debasishg/frdomain）上找到。

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **下载资源**：本书如提供示例代码及资源文件，均可在 下载资源 处下载。
- **提交勘误**：您对书中内容的修改意见可在 提交勘误 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动**：在页面下方 读者评论 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/32392

关于作者

Debasish Ghosh 在领域建模方面有 10 年的工作经验，在过去的 5 年里他主要专注在函数式建模领域。他有着丰富的函数式编程经验，特别在使用 Scala 语言以及 Akka、Scalaz 库方面，这也是本书的基础。他同样是事件源、CQRS 最早的使用者，并在实际应用程序中实践了这些技术。Debasish 还是一本领域相关语言书籍——*DSLs in Action*（www.manning.com/books/dsls-in-action）的作者，该书由 Manning 出版社在 2010 年出版。

关于译者

李源，曾在华为技术有限公司工作8年，经历过开发、SE、PM和PQA等多个岗位，目前在途牛旅游网担任研发总经理一职，是美国质量协会（ASQ）注册质量工程师（CQE）；译者有丰富的开发、架构设计及研发管理经验，先后负责过多个大型项目的方案设计和系统规划，对于C++、Java以及设计模式等领域都有比较深入的研究；曾翻译《Java性能调优指南》一书。

读者可扫码联系译者：

目录

1 函数式领域建模：介绍

本章包括

- 领域模型以及领域驱动设计
- 函数式纯领域模型的好处
- 针对提升响应能力的响应式建模
- 当函数式遇上响应式

假设你正在使用一个大型在线零售商店的门户网站进行购物。在添加完所有要买的商品后，却发现购物车没有记录这些商品的信息，这时会有什么样的感受？或者说在圣诞节前的一个礼拜，你想确认一件商品的价格，但经过无比漫长的等待后才得到响应，你会喜欢这种购物体验吗？这两种场景中应用程序的响应能力都很差。第一个场景描绘了失败响应的缺失——因为一个后端资源的不可用导致整个购物车失效。第二个场景描绘了针对不同负载的响应缺失。可能在节假日这种旺季产生了过度的系统负载，从而使得用户的查询请求响应得无比缓慢。这两种情况都会让用户感到极度沮丧。

对任何你所开发的应用来说，核心概念就是领域模型，它是业务如何进行工作的体现。对于一个银行系统来说，系统功能通常由银行、账号、货币、交易以及报

表等实体组成，它们彼此协同工作并给用户传递一个良好的银行业务体验。模型的责任就是确保用户在使用系统时有一个良好的用户体验。

实现一个领域模型，就是将业务流程翻译成软件。这时需要尝试找到一种方式使得软件与原有业务流程尽可能地类似。为了达到这个目的，会在设计、开发和实现中采用某些技术以及某些范例。在本书中，你会看到如何将函数式编程（FP）与响应式建模结合起来，去实现有良好响应性和伸缩性的模型，同时还要方便管理和维护。本章将介绍这两种范式的基础概念，以及两者如何共同实现模型的响应性。

如果你正在设计开发系统，并且使用某些编程技术，本书能帮助你扩大视野、看到一些新技术，使模型更具有弹性。如果你管理的团队正在开发复杂的系统，使用函数响应式编程会带来很多好处，同时也可以交付更可靠的软件给用户。本书使用 Scala 作为实现语言，但提及的基本概念同样适用于当今行业中的其他语言。

在进入领域建模的核心主题之前，图 1.1 展示了本章为实现领域模型以及函数响应式编程所提供的准备工作。这样可以更容易地了解这些基本概念，到本章结束，就可以用函数式和响应式范式的术语来提炼领域建模技术。

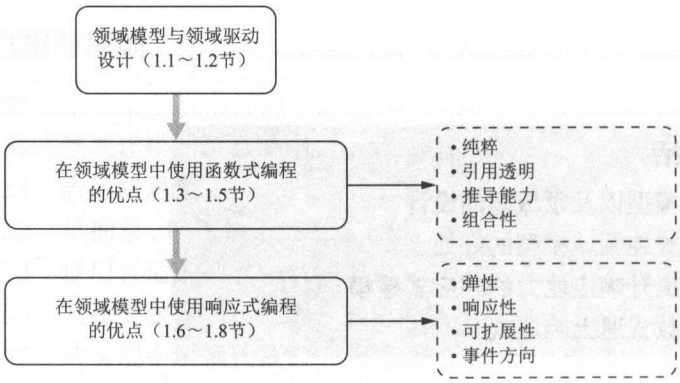


图 1.1 建立函数响应式领域模型。

1.1 什么是领域模型

你最近一次是什么时候从 *ATM* 中取现金、在银行账户中存钱、用在线银行查询工资是否已经转入活期账户，或者向银行索要一份投资报表？所有这些楷体字都与个人银行业务有关，我们称为个人银行业务的领域。领域这个词在这里意味着在业务中的某个具体范围。开发一个系统来自动化银行业务的活动，就是在建立个人银行业务的模型。所设计的抽象、所实现的行为，以及所建立的人机交互界面，这些都反映了个人银行业务，它们在一起组成了领域的模型。

用更正式的语言就是，领域模型是问题领域不同实体之间关系的蓝图以及其他一些重要的细节，比如：

- *属于领域的对象*：在银行领域里，会有诸如银行、账号、交易事务等对象。
- *对象之间互动所展示出来的行为*：比如，在银行系统账户的借方中记入一笔款项，同时生成一个报表给客户。这些就是领域对象之间的典型交互。
- *领域的语言*：对个人银行进行建模时，诸如记入借方、信用、投资组合等术语，或者如“从账户 1 转账 100 美元到账户 2”之类的短语，将会无处不在，然后组成该领域的词汇表。
- *模型操作内的上下文*：其中包含了问题领域相关的假设与约束，并且自动适用于你所开发的软件模型。比如只能为一个活人或实体开放新的银行账号，这一条就可以成为个人银行领域模型的一个上下文约束。

在其他建模练习中，实现一个领域模型最主要的挑战是管理它的复杂度。有些复杂度是问题本身所固有的，无法回避，这些被称为系统的基本复杂度。比如，从银行申请个人借贷，需要根据个人信息决定合适的额度，这就有一个固定的复杂度，由领域的核心业务规则所决定。以上就是一个基本复杂度，它在解决方案模型里是不能回避的。但有些复杂度是由解决方案本身所引入的，比如当实现一个新的银行业务解决方案时引入了额外的负载，如额外的批量处理。这也被称为模型的附带复杂度。

高效模型的实现方式有一个基本原则就是减少附带复杂度的数量。通常情况下，可以使用技术手段来降低附带复杂度的数量，这有助于更好地管理复杂性。比如，如果你的技术可以使模型更好地结构化，那最终实现就不会是一整块庞大而难以管理的软件。它会被分解成多个小型组件，每一个都会结合自身的上下文以及假设来开展工作。图 1.2 描绘了这样一个系统——为了方便只显示了两个组件。但你要知道：对一个模块化的系统，每个组件在功能性上是独立的，而且只通过明确定义的协议与其他组件进行交互。相对于面对一个整体系统，通过这些小模块，可以更好地管理复杂度。

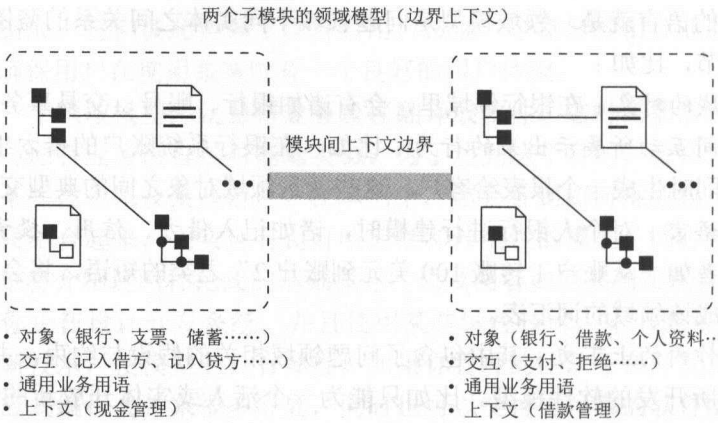


图 1.2 领域模型总览与外部上下文，包括个人银行领域术语。每个小模块都有其自身的假设和业务规则，因此相对于整个大系统来说，这些小模块更容易管理。但需要使用明确定义的协议使小模块之间的通信保持在最小值。

本书会介绍如何使用函数式编程规则并将它们和响应式设计结合起来，进而使领域模型的实现更加容易创建、维护和使用。

1.2 领域驱动设计介绍

在前面章节介绍领域模型时，使用了一些术语，诸如银行、账号、记入借方、记入贷方，等等。所有这些术语都跟个人银行领域有关，而且很容易地表达了它们在业务功能中执行的规则。在为个人银行业务实现领域模型时，如果使用与业务相关的专业术语，对用户来说无疑会更容易理解。比如，模型里可以有一个名为账户的实体，它实现所有行为的变动，类似校验、存款，或者成为货币交易账号。这就是从问题领域（业务）到解决领域（实现）的直接映射。

在实现一个领域模型时，对领域的理解是至关重要的。只有掌握不同实体在现实世界中是如何工作的，才能知道如何在解决方案中实现它们。理解领域并将核心特性抽象成模型的形式，就是我们所说的领域驱动设计（DDD）。Eric Evans 在 *Domain-Driven Design: Tackling Complexity in the Heart of Software*（Addison-Wesley Professional，2003 年）一书中为这个主题提供了绝佳的处理手段。

1.2.1 边界上下文

1.1 节描述了模型的模块以及模块化给领域模型带来的一些好处。任何复杂的领域模型都是一系列小模型的集合，每个小模型都有它自己的数据和领域术语表。在领域驱动设计的世界里，术语边界上下文就描述了在整个模型里的某个小模型。

于是，完成的领域模型就是多个边界上下文的集合。让我们看一个银行业务系统：投资组合管理系统，税收与调整报告和定期存款管理就可以被拆分成不同的边界上下文。边界上下文通常在一个非常高水平的粒度上描述系统里某个完整的功能域。

但是，在完整领域模型中拥有多个边界上下文时，它们之间如何进行通信？要记得，每个边界上下文都是独立的模块，但可以跟其他所有边界上下文发生交互。当设计模型时，这些通信都被实现为特定的服务或接口集合。后面会看到这种实现。基本思路是将这些交互控制在尽可能小的数量，这样每个边界上下文就能充分地内聚，同时降低与其他边界上下文的耦合。

在下一节中会看到每个边界上下文的内容有什么，并学习到一些领域建模的基本要素，这些将组成模型的核心。

1.2.2 领域模型元素

不同类型的抽象定义了领域模型。如果有人要我们列出一些个人银行领域的元素，我们就有机会自己命名这些内容，比如银行和账户，账户类型诸如核验、存款和货币市场，以及交易类型，如记入贷方、记入借方。但很快就会发现很多元素在创建方法上是非常相似的，通过业务管道处理，并最终从系统中排出。我们来看一个例子，如图 1.3 所示，这是一个客户账户的生命周期。每个客户账户都会被银行创建出来并传递一系列状态，以此作为银行、客户或其他外部系统的行为的结果。

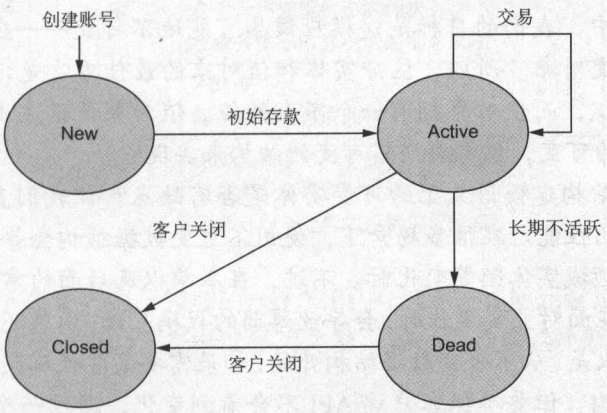


图 1.3 客户账户的生命周期状态，从一个状态转移到另一个状态依赖于在前一个状态上执行的行为。

每个账户都有一个身份，在账户的整个生命周期内，系统会对它进行管理，这类元素被称为实体。对于一个账户来说，它的身份就是它的账户号码。账户的很多属性在系统的生命周期内都有可能发生变化，但开户时生成并分配给账户的号码总是确定的。两个同时生成并有相同属性的账户被认为是不同的实体，因为账户号码

是不同的。

每个账户可能会有一个地址——账户持有人的居住地址。一个地址由其包含的值唯一定义。一旦改变地址的任意属性，它就变成了一个不同的地址。我们能否从语义上区别账户和地址的不同之处？一个地址没有任何身份，它完全取决于它所包含的值。我们称这种对象为值对象。实体与值对象的另一个区别是值对象是不可变的——在创建值对象之后，不能在不改变对象的情况下改变它的内容。

实体与值对象的区别是领域建模中一个最基本的概念，我们必须对它有清楚的认知。当谈到一个账户时，指的是一个特定的账户实例，包含一个账户号码、持有人姓名以及其他属性。这里某些属性组合起来形成这个账户的独特身份标识。例如，账户号码就是这个账户的身份属性，哪怕有两个有相同属性值的账户（比如持有人姓名或开户时间），但如果账户号码不同，那它们就是两个不同的账户。一个账户就是一个有特定身份的实体，但对于一个地址只需要考虑值的部分。因此在模型中，可以选择对一个特定地址只生成一个实例，然后在所有居住在这个地址的持有人中间共享它。这只跟值有关系。一个实体的某些属性的值可以改变，但是身份标识不会改变。比如可以改变一个账户的地址，但它还是会指向相同的账户。但不能改变值对象的值，否则它就变成了另外一个值对象。所以值对象是不可变的。

实体和值对象的“不变性”

在本章后面讨论实现方式时，对实体和值对象的不变性会有不同的看法。在函数式编程中，我们的目标是建模应该尽可能地不可变——应该将实体尽可能建模为不可变对象。所以，区分实体和值对象的最佳方法是记住实体有一个不变的身份标识，而值对象拥有一个不变的值。值对象是语义上的不可变，而实体是语义上的可变，但需要用不可变的结构来实现它。

用不可变结构建模语义上的可变实体是否有缺点？让我们直面它——可变引用拥有更好的性能。在很多场景下，使用不可变数据结构会导致更多的对象，特别是当一个领域实体频繁变化时。不过，在本章以及后面的章节里可以看到，可变数据结构在面对并发操作时，会导致薄弱的代码基础，同时还非常难以理解。所以通常的建议是，从不可变数据结构开始，如果需要某些代码获得更好的性能，再使用可变结构。但要确保客户端API不会看到变化，通过一个引用包装函数来封装易变部分。[1]

1 例如Scala的Collections API的实现。比如`List::take`或者`List::drop`，在封装内部使用了可变结构，但客户端API看不到。对于调用，客户端会取回一个不可变的`List`。

任何领域模型的核心都是不同领域元素之间行为或交互的集合。相较于单独的实体或值对象，这些行为位于一个更高级别的粒度。我们认为它们是模型给出的概念服务。让我们看一个银行系统的例子。比如一个用户去银行或用 ATM 机在两个账户之间进行转账。这个行为会导致一个账户中减少一笔款项并在另一个账户中存入这笔款项，这也反映为不同账户里的余额变化。而且必须做验证，比如，账户是否处于激活状态，源账户是否有足够的资金进行转账。在每个这样的交互中，会涉及非常多的领域元素，包括实体和值对象。在 DDD 中，将这种行为的集合建模为一个或多个服务。依靠架构和模型特定的边界上下文，既可以将它打包成一个独立的服务（可以命名其为 `AccountService`），也可以将其作为一个服务集合的一个组成部分，这个集合是一个更通用的模块，被命名为 `BankingService`。

区分领域服务和实体、值对象的主要方式是粒度的层次。在一个服务里，多个领域实体根据特定业务规则进行交互的同时执行系统中特定的功能。从实现的角度来看，一个服务是一系列领域实体及值对象功能行为的集合。它囊括了一个完整的业务操作，同时对于用户或银行来说它有一个具体的值。表 1.1 汇总了到此为止所提到的 3 个重要的领域元素的特性。

表 1.1 领域元素

元　素	特　性
实体	有一个身份标识 在生命周期里传递多种状态 在业务中通常有明确的生命周期
值对象	不可变 能在实体间自由共享
服务	比实体和值对象更高级别的抽象 涵盖多个实体和值对象 通常对一个业务的用例进行建模

图 1.4 描述了在个人银行领域的一个案例中，这 3 种领域元素是如何关联起来的。这是 DDD 的一个基础概念，在继续学习之前，请确保理解这个基本知识。

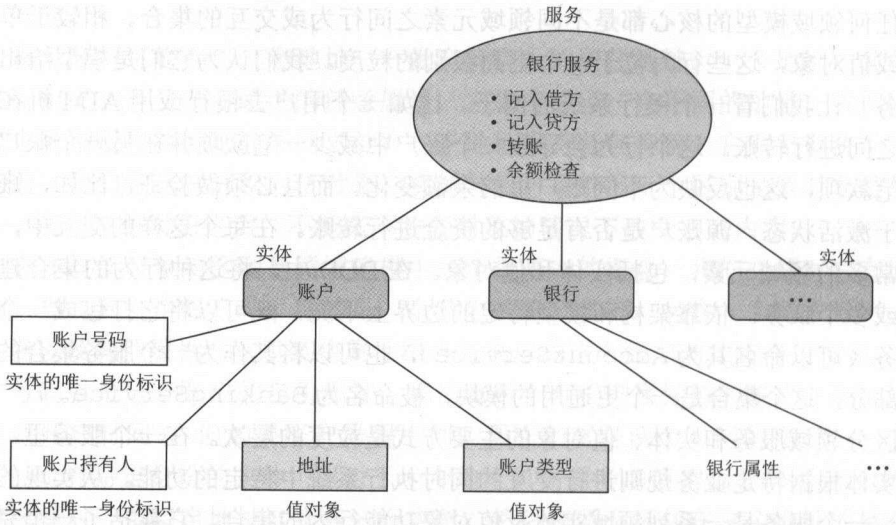


图 1.4 模型的领域元素关系图。这个例子来源于个人银行领域。注意，账户、银行等都是实体。一个实体可以包含其他实体或值对象。服务位于更高水平的粒度，执行的行为也涵盖多个领域元素。

领域元素语义与边界上下文

让我们用一个很重要的概念来结束不同领域元素的讨论，这就是将它们的语义关联到边界上下文。当我们说地址是一个值对象时，它只是在定义边界上下文范围内的一个值对象。在个人银行业务应用的边界上下文里，地址可能是一个值对象，也不需要通过它们的标识来追踪。但如果是另外一个实现地理编码服务的边界上下文，就需要通过经纬度来追踪地址，并且每个地址都可能必须打上唯一的 ID 标识。在这个边界上下文中，地址就成为了一个实体。类似地，在个人银行业务应用中，账户可能是一个实体，但在投资报告的边界上下文中，账户可能仅仅只是信息的一个载体，只需要被打印出来，因此它被实现为一个值对象。领域元素的类型通常跟定义它的边界上下文有关。

1.2.3 领域对象的生命周期

在任意模型中的每个对象（实体或值对象）都必须有明确的生命周期模式。在模型中的每个对象类型，必须通过定义途径来处理下列每个事件。

- *创建*：在系统中对象如何被创建。在银行业务系统中，可能需要一个特殊的抽象来负责创建银行账户。
- *参与行为*：当对象在系统中交互时，它在内存中如何被表现出来。这也是在系统中建立一个实体或者一个值对象的方式。一个复杂实体可能会包括其他实体以及值对象。比如在图 1.4 中，账户实体会包含对其他实体如银行或其

他值对象（如地址或账户类型）的引用。

- 持久化：对象如何在持久化的形态下维护。这包含一系列问题，比如：如何将元素写入持久化存储；如何检索详情来响应系统的查询；以及如果持久化方式是关系型数据库，那么如何插入、更新、删除，或者查询一个实体，如账户。

通常情况下，需要一个统一的词汇表。在下面的章节中会使用特定的术语来描述如何在模型中处理这 3 种生命周期事件。我们称其为模式，因为会在领域建模的不同上下文中反复使用它们[1]。

工厂

当我们拥有一个复杂的模型时，使用专门的抽象来处理其生命周期的各个部分一向是一种很好的实践。它再不是一堆杂乱的代码，包含了一个个创建实体的代码片段，而是用一个模式将它们集中起来。这种策略服务于以下两个目的。

- 将所有创建的代码保存在一个位置。
- 抽象了来自调用者创建实体的过程。

比如说，有一个账户工厂（factory），它获取创建一个账户所需的不同参数，然后交给你一个新创建的账户。从工厂取回来的账户可能是一个校验、存款或者货币市场账户，这依赖于传入的参数。所以工厂使我们可以使用相同的 API 创建不同类型的对象。它抽象了创建对象的过程以及类型。

创建逻辑被隐藏在工厂内部。但工厂属于哪里呢？毕竟，工厂提供了一个服务——创建以及可能的初始化服务。这里工厂的职责就是提供一个完整构造的、最低程度的、合法的领域对象实例。一个选项就是使工厂成为定义领域对象模块的一个组成部分。在 Scala 中使用伴生对象（companion object）就有天然的实现，在清单 1.1 中会有描述。另一个选项是将工厂作为领域服务集合的一个组成部分。第 2 章会详细讲述一个这样的实现。

清单 1.1 在 Scala 中用工厂实例化账户

```
trait Account {                          <-- 账户实体接口与不同类型的账户
  //..
}
case class CheckingAccount(/* parameters */) extends Account
case class SavingsAccount(/* parameters */) extends Account
case class MoneyMarketAccount(/* parameters */) extends Account
```

1 使用术语模式会有一点点不精确的感觉，因为可能不会完全符合GoF所遵循的严格框架（*Design Patterns: Elements of Reusable Object-Oriented Software*，作者：Erich Gamma等人，Addison Wesley Professional, 1994年）。我们称*factory*、*repository*与*aggregate*为领域生命周期模式，这是Eric Evans在他的*Domain-Driven Design*一书中所使用的专业术语。

```
object Account {
  def apply(/* parameters */) = {
    // instantiate Checking, Savings or MoneyMarket account
    // depending on parameters
  }
}
```

Scala 里的伴生对象包含了 factory

工厂方法用来实例化账户

聚合

在前面的个人银行模型里，账户可以被认为是一组相关对象的组合。它一般包括下面这些内容。

- 核心账户标识属性，如账户号码。
- 各种非标识类属性，如持有人姓名，账户开户日期以及关闭日期（如果它是一个被关闭的账户）。
- 对其他对象的引用，比如地址和银行。

通过把账户看成一个内部由一致性边界组成的整体，就可以在脑海中想象整个对象图。在实例化一个账户时，所有这些独立参与的对象以及属性必须与领域的业务规则保持一致。不可能让一个账户在有开户日期之前先有关闭日期，也不可以让一个账户里没有任何持有人姓名。这些都是合理的业务规则，实例化的账户必须让所有组合的对象遵守每一条规则。如果在图中识别出参与对象的集合，那么这张图就成为了一个聚合（aggregate）。一个聚合可以由一个或多个实体、值对象以及原始属性（primitive attribute）组成。除了确保与业务规则的一致性，边界上下文里的聚合也通常被看作模型中的执行边界。

聚合里的一个实体构成聚合根。在一定程度上来说它有点像整个图的守门人，同时也是聚合与客户进行交互的单一点。聚合根有两个目标需要关注：

- 确保聚合内部业务规则与执行的一致性边界。
- 防止聚合的实现泄露给它的客户端，聚合支持的所有操作都要通过外观执行。[1]

清单 1.2 显示了用 Scala 设计一个账户聚合。它包含聚合根账户（同样也是一个实体），包含实体如银行，以及值对象如地址，这些都是它的组成元素。[2] 设计聚合不是一件容易的事，特别是对于 DDD 的初学者。除了 Eric Evans 的 *Domain-Driven Design* 之外，还可以看一下 Vaughn Vernon 的文章 *Effective Aggregate Design*，这里面讨论了 3 个需要关注的内容，有助于设计一个好的聚合（http://dddcommunity.org/library/vernon_2011/）。

1 这里可以参考外观模式，即对外交互的单点化。——译者注

2 现实中设计聚合时会发现，考虑到操作性能和一致性，必须从聚合中优化掉许多组合的实体，并只保留根和值对象。比如，你会选择保留一个银行ID而不是整个Bank实体作为Account聚合的组成部分。

清单 1.2 账户聚合

```
trait Account {
  def no: String
  def name: String
  def bank: Bank
  def address: Address
  def dateOfOpening: Date,
  def dateOfClose: Option[Date]
  //..
}

case class CheckingAccount(
  no: String,
  name: String,
  bank: Bank,
  address: Address,
  dateOfOpening: Date,
  dateOfClose: Option[Date],
  //..
) extends Account

case class SavingsAccount(
  //..
  rateOfInterest: BigDecimal,
  //..
) extends Account

trait AccountService {
  def transfer(from: Account, to: Account, amount: Amount): Option[Amount]
}
```

账户聚合的基本特征。

对另外一个对象的引用。

地址是一个值对象。

账户的具体实现，注意这里的字段覆盖了 trait（特质）里的 def。

Scala 里的 case class

清单 1.2 中使用 Scala 的 case class 对账户聚合进行了建模。在指南中 Scala 的 case class 提供了一个便捷的方式来设计不可变对象。默认情况下，这个类获取的所有参数都是不可变的。因此，使用 case class，是一个更容易设计聚合的捷径，当然使用不变性所带来的好处同样重要。

清单 1.1 和 1.2 使用了 Scala 里的 *trait*。*trait* 可以使用基于 mixin 的方式组合在一起，所以可以使用它来定义模块。*mixin* 是一种小型抽象，可以与其他组件混合在一起形成更大的组件。

关于 case calss、mixin 以及 trait 的细节，可以查看 Scala 的官方主页（www.scala-lang.org），或者参考 Martin Odersky 的 *Programming in Scala*（Artima Press, 2016 年）一书。

前文首先用 Scala 的 trait 实现了账户聚合的基本特征，然后用 case class 的方式进行了扩展。正如前文所述，可以很便捷地使用 case class 来建立不可变数据结构模型。这也是我们所说的*代数数据类型*，在后文会讨论更多的细节。但让我们先来看一个前面接触到的账户实体聚合的概念。

在 1.2.2 节中，提到我们可以更新一个实体的某些属性但不改变它的身份标识。这意味着实体是可以更新的。但在这里，我们将账户实体建模为一种*不可变的*抽象。这看起来似乎是一个很明显的矛盾，但事实并非如此。我们允许更新实体，但仅限于*不能在适当的地方*使对象保持可变的函数方式。与改变对象本身相反，更新会使要变更的属性值生成一个新的实例。[1] 这样做的好处是可以继续将原来的抽象共享为一个不可变实体，同时根据这个实体生成一个新的实例并进行更新。从函数式的角度来考虑，我们将尽可能保证实体的不变性（与值对象很类似）。同时这也是指导模型设计的原则之一。清单 1.2 同样也展示了一个领域服务（AccountService）的案例，其中使用了账户聚合来实现两个账户之间的资金转移。

仓储

现在我们知道，聚合是通过工厂创建出来的，并在对象生命周期的激活阶段在内存里代表基础的实体（如图 1.3 所示，回顾一个账户的生命周期）。但还需要一个途径去保存一个聚合，当我们不再需要它时，不能把它直接抛弃，因为后面可能还会因为其他目的而需要它。

仓储（repository）提供了一个接口，以持久化的形态来存放这个聚合，这样当我们需要它时，就可以把它取回来变为内存中的实体形态。通常来说，仓储的实现基于持久化存储，比如一个关系型数据库管理系统（RDBMS），尽管并不是必须这样。[2] 要注意的是，聚合的持久化模式可能会与内存中聚合的表现完全不同，而且通常被底层存储数据模式所驱动。仓储的职责（参见清单 1.3）就是提供在持久化存储中操作实体的接口，但不会暴露底层关系型数据模型（或任何底层存储支持的模型）。

清单 1.3 AccountRepository——在数据库中操作账户的接口

```
trait AccountRepository {
  def query(accountNo: String): Option[Account]
  def query(criteria: Criteria[Account]): Seq[Account]
  def write(accounts: Seq[Account]): Boolean
  def delete(account: Account): Boolean
}
```

1 如果你没有耐心，就跳到清单1.4，在那里debit和credit方法会用更新的余额数目来创建新的Account实例。

2 在许多小型应用中，可以使用内存中的仓储，但这种场景并不多见。

这个仓储的接口不包含任何底层持久化存储的相关特性。它可能是一个关系型数据库，也可能是一个 NoSQL 数据库——这只有在具体执行时才知道。聚合在内存中生成一个实体实例，仓储就会在持久化存储中保存一个一模一样的实体。聚合隐藏了对象在内存中的底层细节，而仓储抽象了对象在持久化存储中的底层细节。在清单 1.3 中可以看到 AccountRepository 从底层存储操作账户，清单中没有展示任何仓储的具体实现。用户依然可以通过一个聚合与仓储进行交互。我们看看下面的思路就知道聚合是如何为实体的整个生命周期提供一个单一窗口的：

- 提供一串参数给工厂，并取得一个聚合（如 Account）。
- 将这个聚合（清单 1.2 中的 Account）作为你的协议，对应通过服务（如清单 1.2 中的 AccountService）实现的所有行为。
- 通过聚合将实体在仓储中持久化（清单 1.3 的 AccountRepository）。

现在已经了解了在模型中模块化的几个手段：使用边界上下文，需要实现的领域元素 3 个最重要的类型（实体、值对象、服务），以及操作它们的 3 种模式（factory、aggregate、repository）。现在我们必须意识到，这 3 种元素类型在领域中的交互（诸如银行系统中的 debit、credit 等）以及它们的生命周期都是通过 3 种模式来控制的。在领域驱动设计中要讨论的最后一件事就是将它们捆绑到一起，这被称为模型的词汇表，在下一节中会学到它为什么很重要。

1.2.4 通用语言

现在我们已经拥有组成模型的实体、值对象和服务，也知道所有这些元素都需要彼此进行交互，以便实现业务执行的不同行为。作为一个具有工匠精神的软件工程师，对交互的建模不应该仅仅被底层硬件所理解，同时还要满足人们的好奇心。这个交互需要体现出基本的业务语义，并且包含正在建立的问题领域的词汇表。所谓词汇表，就是在用户场景中所涉及的对象与行为的名称。在案例中，实体如 Bank（银行）、Account（账户）、Customer（用户），行为如 debit（记入借方）和 credit（记入贷方），都与业务术语强相关，因此也成为了领域词汇表的一部分。对词汇表的使用需要对业务进行更大范围的抽象，从而形成更小的词汇表。比如，可以这样实现一个 AccountService（领域服务）：[1]

1 如果不理解实现的细节也没有关系。在本章后面讨论函数式领域建模的演变时会实现这个服务。

```
trait AccountService {
  def debit(a: Account, amount: Amount): Try[Account] = //..
  def credit(a: Account, amount: Amount): Try[Account] = //..

  def transfer(from: Account, to: Account, amount: Amount) = for {
    d <- debit(from, amount)
    c <- credit(to, amount)
  } yield (d, c)
}
```

让我们更仔细地看一下这个实现是如何体现前面所说的易懂性的。

- 函数体最小化，同时不包含任何无关的细节。它仅仅封装了涉及两个账户间转账的领域逻辑。
- 该实现使用了银行领域的术语，这样一个熟悉业务领域但对底层实现平台一无所知的人同样可以很容易地理解它能做什么。
- 该实现只描述了正常的执行路径。所有的异常路径都通过抽象被封装起来了。如果了解 Scala 就会知道，这里使用的 for 表达式是单子化的（monadic），它可以在序列执行过程中照顾好所有异常。[1] 后面还会讨论更多有关这方面的内容。

Eric Evans 将它称为*通用语言*（ubiquitous language）。在模型中使用领域词汇表并用术语进行互动，就如同领域所说的语言一样。从正确命名实体与原子行为开始，将词汇表扩展到更广泛的抽象。不同的模块可以使用不同的“方言”，在不同的边界上下文中相同的术语可能有不同的含义。但在一个上下文内部，词汇表必须是清晰明确的。

要形成一个一致的通用语言，很多方面与设计合适的模型 API 有关。API 必须具备足够的表达力，这样一个该领域的专家可以只看 API 就能理解上下文。这也是我们所知道的*领域特征语言*，在我写的 *DSLs in Action*（Manning，2010 年）一书中可以看到这方面更多的细节。

1.3 函数化思想

领域建模有很多种途径，但在过去的这十几年中，面向对象技术在复杂领域模型的开拓中占据了绝对的主导地位。本书中，我会有一点点激进，会用简单的*老旧函数*作为领域行为建模的主要抽象方式。在接下来的章节中，可以从建模以及维护软件两个方面，看到这样做的好处。

1 在Scala中，for表达式可以很简单地与map/flatMap/filter操作配合使用。更多细节可以参考http://docs.scala-lang.org/tutorials/tour/sequence-comprehensions.html。

> 有些时候，优雅的实现仅仅是一个函数。不是一个方法，不是一个类，不是一个框架，只是一个函数。
>
> ——John Carmack
>
> （https://twitter.com/ID_AA_Carmack/statuses/53512300451201024）

不过还是让我们从近些年所使用的范式作为开始。在这个例子中，将深入分析一个实现，然后逐步将其改变为函数式的变体。让我们回到日常生活中都会遇到的一个领域：个人银行业务。这个简单的模型会包括：一个聚合 Account、一个值对象 Balance、一些其他属性，以及 debit 和 credit 这对操作，如清单 1.4 所示。

清单 1.4 个人银行业务领域模型实例

```
type Amount = BigDecimal

case class Balance(amount: Amount = 0)

class Account(val no: String, val name: String, val dateOfOpening: Date) {
  var balance: Balance = Balance()

  def debit(a: Amount) = {
    if (balance.amount < a)
      throw new Exception("Insufficient balance in account")
    balance = Balance(balance.amount - a)
  }

  def credit(a: Amount) = balance = Balance(balance.amount + a)
}

val a = new Account("a1", "John")
a.balance == Balance(0)

a.credit(100)
a.balance == Balance(100)

a.debit(20)
a.balance == Balance(80)
```

聚合

聚合中的可变状态

操作可变状态

如果等式条件满足，断言就返回 true

清单 1.4 中的内容浅显易懂。类 Account 包含一个可变状态，账户用它来保存余额。方法 debit 和 credit 在任何时候都可以及时地通过直接改变对象的状态来改变账户余额的值。

测验时间 1.1 这种模型最主要的缺点是什么？

认真想一想，回顾一下清单 1.4。我们将要讨论的内容很可能是为什么要重视函数化思想与建模的最重要的原因。

测验答案 1.1 主要问题是易变性，它会从两个方面打击我们：很难在并行设置下使用抽象，而且很难推理代码。

这里要解释得更详细一些。`var balance:Balance` 在领域模型中是一个可变状态。这里的关键词是“可变”，这就意味着该对象包含的这种状态可以被对象的多个客户端更新。进一步延伸到一个并发环境中，在决定状态值的时候就会遇到各种各样的矛盾类型，在任何时间任何位置。这本身就是一个非常庞大的话题，在 Brian Goetz 所著的 *Java Concurrency in Practice*（Addison-Wesley Professional, 2006 年）一书中可以获得所有此类问题的详细描述。可变状态在推导代码时也是一个反模式，本章的后面会有这块内容。[1] 尽管它看上去是一个有说服力的建模手段，但可变状态带来的问题远远多于它解决的问题。我们需要想办法干掉这些可变状态。

接下来看看是否可以继续用面向对象的思想来解决前面代码的这个主要缺陷。清单 1.5 展示了下一步的尝试，针对清单 1.4 中在模型中引入可变状态造成的“罪恶”进行弥补。

清单 1.5 不可变 Account 模型

```
type Amount = BigDecimal

case class Balance(amount: Amount = 0)

class Account(val no: String, val name: String,
  val dateOfOpening: Date, val balance: Balance = Balance()) {

  def debit(a: Amount) = {
    if (balance.amount < a)
      throw new Exception("Insufficient balance in account")
    new Account(no, name, dateOfOpening, Balance(balance.amount - a))
  }

  def credit(a: Amount) =
    new Account(no, name, dateOfOpening, Balance(balance.amount + a))
}

val a = new Account("a1", "John", today)
a.balance == Balance(0)

val b = a.credit(100)
a.balance == Balance(0)
b.balance == Balance(100)

val c = b.debit(20)
b.balance == Balance(100)
c.balance == Balance(80)
```

Balance 现在是不可变的

方法 debit 和 credit 将创建 Account 新的实例

行为不变性——这个地方账户余额不能再被改变了

现在可变状态没有了！Account 的每次操作都会用变更后的状态生成一个新

1 如果“推导代码”听起来有点陌生，没关系，我们很快就会学到更多相关内容。

的对象。不用再包含一个可变状态，这个新的 Account 类可以自己承载状态。一旦生成一个该类的实例，在实例内部就有 balance 作为状态。区别在于这个状态是不可变的。如果不创建另一个 Account 对象，就不能改变它的值。这也正是 debit 和 credit 操作所要做的事。Scala 会确保传递给类构造器的参数默认是不可变的。可以选择通过将其定义为 var 使之成为可变的。但 var 是一个非常露骨的修饰词，因为需要去申请以获取可变性——而更好的决定是鼓励不可变抽象设计。

现在我们已经将 Account 改造成一个不可变抽象，可以在并发配置下通过线程自由地共享 Account。[1] 这是一个巨大的收获，也是向函数化思想迈出的第一步：用纯函数的方式工作，接受输入然后生成输出，不再依赖或影响可变状态。不变性在这里扮演一个非常重要的角色。

但这还没有结束。Account 依然是一个同时拥有状态与行为的抽象。后面会看到，我们的想法是将这两者进行解耦，这会带来更好的模块化，也因此有了更好的可组合性。不过在此之前，先看一下，如果用纯函数建模，代码会得到什么好处。

1.3.1 哈，纯粹的乐趣

想象一下，如果回到学校时光，然后尝试从数学的角度学习“函数”的定义。现在讨论函数式编程，那么这个函数和我们在数学班里所学的函数有什么区别？

> 在数学里，一个函数就是一组输入与一组容许的输出之间的关系，并且每个输入都与唯一一个输出相对应。
>
> ——维基百科，http://en.wikipedia.org/wiki/Function_(mathematics)

这个定义永远不会提到涉及共享可变状态的函数。函数的输出纯粹由输入决定——如图 1.5 所示，将函数（f）建模成一个黑盒，输入（x）得到输出（y）。在函数式编程中，需要努力使函数像数学函数一样。

测验时间 1.2 清单 1.4 与 1.5 中，哪个模型看起来更接近前面介绍的函数的定义？

现在我们已经了解了函数的定义，而且也知道了为什么要在领域模型中尝试着达到同样的效果。要回答这个问题根本就是小意思。哪个模型对可变状态依赖更少，谁就更接近于纯粹的数学函数。

---

1 这里只讨论关于共享Account对象。如果想要在一个执行里组合多个debit和credit，还需要掌握原子性。

测验答案 1.2 清单 1.5 因为对 Account 做了不可变抽象，因此它更接近该定义。

在图 1.5 中，模型 y=f(x) 假设 f 是一个平方函数。那么不论调用多少次 f，square(3)=9 这个结果是绝对相同的。接下来结合刚才介绍的两种模型针对这个问题做更多细节方面的讨论。

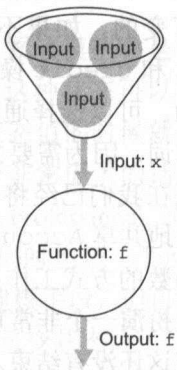


图 1.5 y=f(x) 建立了一个纯函数模型。f 是一个黑盒，输入一个 x，得到一个输出 y。

在清单 1.4 中的可变模型中，Account 对象调用了 debit(100) 并得出一个值，但这个结果不仅仅依赖输入参数 100 以及对象本身（可以把它当作一个隐性参数），还依赖于共享该对象的其他客户端。因为共享 Account 对象的所有客户端都有同样的通道访问可变状态。这也是与纯函数的区别所在。

在清单 1.5 中的不可变模型中，Account 对象本身就包含当前状态。因此，对一个当前余额是 2000 的 Account 对象的调用 debit(100)，将会生成一个余额更新为 1900 的新的 Account 对象。输出仅仅依赖于提供的输入，所以这个模型拥有数学函数的纯洁性。

我们很快就会看到，Account 的不可变模型其实是一个函数式模型的面向对象版本。它还将函数建模为一个类的方法。在这种情况下，通常会使我们左右为难，哪个函数应该成为哪个类的组成部分。同样，要将这些组合成为不同类方法的函数也会非常困难。

在这个例子里，debit 和 credit 是针对单个账户的操作，将它们看成 Account 的行为。但诸如 transfer 这种操作会涉及两个账户。那它应该成为类 Account 的一部分，还是应该变成领域服务的一部分呢？应该如何处理针对一个账户的其他服务，比如每日余额结算或利息计算？我们可能会试着将它们放到一个类里形成一个臃肿的抽象。但是，将这类行为放进一个特定的聚合会妨碍模块化与组合性。所以，当设计函数式领域模型时，需要遵守一些通用原则：

- 将不可变状态建模为*代数数据类型*（algebraic data type，ADT）。
- 在模块中将行为建模为函数，这里的模块是指一个粗糙的业务功能单元（比如一个领域服务）。这样，就将状态从行为中分离了出来。行为比状态更好组合，因此，在模块中包含相关的行为有助于提升组合性。
- 记住，模块里的行为对 ADT 中的类型起作用。

测验时间 1.3 对或错：面向对象范式[1]捆绑了状态和行为。函数式编程则对它们进行了解耦。

1 主要用主流的OO语言实现。

要回答上面的问题，先看一下如何用函数式 Scala 实现 Account 模型。清单 1.6 就是对前面的实现进行了改进的模型。它包含了一点点 Scala 构造，不过可以暂且忽略它们。但这里面来自函数化思想的结论，需要应用到领域模型中去：

- 在 Scala 中，用 case class 对 ADT 建模。默认情况下，所有 ADT 的参数都是不可变的，这也就意味着不需要特定的机制来保证模型的不变性。
- ADT 的定义不包含任何行为。可以注意到 debit 和 credit 现在是在被定义为一个领域服务的 AccountService 里。[1] 服务在模块中定义，在 Scala 中被实现为 trait。trait 充当 mixin（混入类），可以很容易地把多个小模块组合成大模块。当需要一个模块实例时（上下文中的一个服务），请使用关键字 object。就像前面所说的，要用函数化思想将状态和行为解绑——状态现在定居在 ADT 里，而行为被建模为模块里的独立函数。
- debit 和 credit 都是纯函数，因为它们没有被绑定到任何特定对象。它们获取参数，执行一些函数功能，然后生成特定的输出，就像图 1.5 中的 y=f(x) 模型。
- 清单 1.6 中引入了一些其他构造如 Try、Success 和 Failure，相对抛出异常来说，这会更加函数化，而且有更好的组合性。在后面的贴士“Scala 里的异常”中会介绍在 Scala 中如何处理异常。后面的章节中也包含这个专题，以及更多关于函数式编程模式的细节。

清单 1.6 净化模型

```
import java.util.{ Date, Calendar }
import scala.util.{ Try, Success, Failure }

def today = Calendar.getInstance.getTime
type Amount = BigDecimal

case class Balance(amount: Amount = 0)

case class Account(no: String, name: String,
  dateOfOpening: Date, balance: Balance = Balance())

trait AccountService {

  def debit(a: Account, amount: Amount): Try[Account] = {
    if (a.balance.amount < amount)
      Failure(new Exception("Insufficient balance in account"))
    else Success(a.copy(balance = Balance(a.balance.amount - amount)))
  }
```

Account 聚合现在是一个 ADT

包含 debit 和 credit 操作的领域服务

1 如果已经忘记了领域服务，可以回顾一下1.2.3节。

```
  def credit(a: Account, amount: Amount): Try[Account] =
    Success(a.copy(balance = Balance(a.balance.amount + amount)))
}

object AccountService extends AccountService

import AccountService._

val a = Account("a1", "John", today)
a.balance == Balance(0)

val b = credit(a, 1000)
```

用关键字 object 实例化服务

纯函数调用，返回 Try[Account]

图 1.6 汇总了用 Scala 将面向对象的不可变领域模型转变为函数式变体所做的改动。

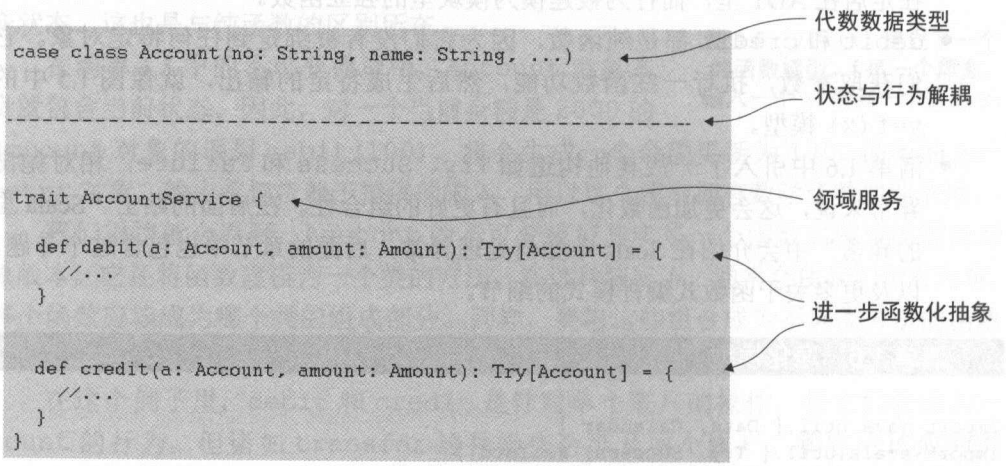


图 1.6 从面向对象不可变建模到函数化抽象。要注意到，我们已经将状态和行为分隔开。状态被编码为代数数据类型 Account，而行为都被包含进一个领域服务。诸如 Try 这样的构造也有助于建立可组合的抽象。

测验答案 1.3 主流的面向对象语言鼓励函数和状态封装在同一个抽象里。在 OO 语言里这个抽象就是 class。

下一节主要谈函数组合。一起窥探另一个很酷的组合效果，比如用 Try 重构为函数式抽象带来的结果。现在可以组合多个 debit 和 credit，就像下面这样：

```
val a = Account("a1", "John", today)

for {
  b <- credit(a, 1000)
  c <- debit(b, 200)
  d <- debit(c, 190)
} yield d
res5: scala.util.Try[Account] = Success(Account(a1,John,Sat Nov 22
 02:38:03 GMT+05:30 2014,Balance(610)))
```

Scala 里的异常

在函数式编程里，异常被认为是不纯粹的。为了能函数化地处理异常，Scala 定义了一个抽象 util.Try，它包含 Success 和 Failure 两个具体实现。清单 1.6 用这个抽象来捕获 generateAuditLog 操作可能产生的所有异常。注意，generateAuditLog 函数取得一个账户和一个数目，然后尝试生成一个字符串型的审计记录。Try[String] 作为返回类型意味着这个操作可能失败，如果失败的话将返回一个 Failure。现在不需要理解太多其中的细节，只要记得 Try 是一个可组合的抽象，而且可以和其他抽象用纯函数的方式进行组合。

1.3.2 纯函数组合

什么是函数组合？在回答这个问题之前，先看一下组合的定义：

> 将事物安放或安排在一起的方式：零件或元素的结合形成新的事物。
>
> ——Merriam Webster（www.merriam-webster.com/dictionary/composition）

组合就是将不同部分捆绑在一起形成一个整体。在数学上，它就是一个函数与另一个函数生成第三个函数的逐点应用。这里有一点点隐藏的创新——创造了一个包含两种函数功能的全新函数。比如有两个函数：f:X->Y 和 g:Y->Z，将这两个组合在一起得到一个新的函数，将 x 代入 X，通过 g(f(x)) 得到 Z。

让我们把这个案例翻译成函数式编程的领域。假设有一个函数，square:Int->Int，输入一个整数作为参数，生成另外一个整数，值为输入的平方。同时，还有另外一个函数，add2:Int->Int，它输入一个整数并将其加 2。将这两者的组合定义为 add2(square(x:Int))，也就是在输入的整数平方结果上再加 2。

这就是我们首先要意识到的，函数式编程是基于函数组合的，这和在数学上处理函数非常类似，也就是常说的函数式编程的组合性。

测验时间 1.4 假设有两个函数：f:String->Int 和 g:Int->Int，会怎么定义 f 和 g 的组合呢？能想到哪些满足这个组合的真实函数？

利用组合性的特质，可以用小函数构成一个大函数。本书的一个重要主题就是探索不同的方式来组合各种函数。我们会使用 Scala，它完全具备把这种组合变得更加简单的能力。关于 Scala 中函数式编程的更多细节，请参考 Paul Chiusano 和 Runar Bjarnason 的著作 *Functional Programming in Scala*（Manning，2014 年）。

在这本书中可以看到用 Scala REPL[1] 来组合函数的例子，REPL 是与 Scala 解释器交互的一个环境。

测验答案 1.4 组合可以被定义为 g(f(x: String))。实际的例子就是：把 f 作为计算一个字符串长度的函数，把 g 作为将输入整数翻倍的函数。于是，double(length(x:String)) 就是组合两个函数的实际例子，它会返回输入字符串长度两倍的值。

现在我们已经熟悉了函数式编程的基本技术，是时候更进一步了。到现在为止，我们所讨论的组合，都是把一个个独立的函数串联到一起，将一个函数的输出作为另一个函数的输入。但在讨论函数式编程里的组合特性时，就远远不止这些内容了。让我们看如图 1.7 所示的例子。

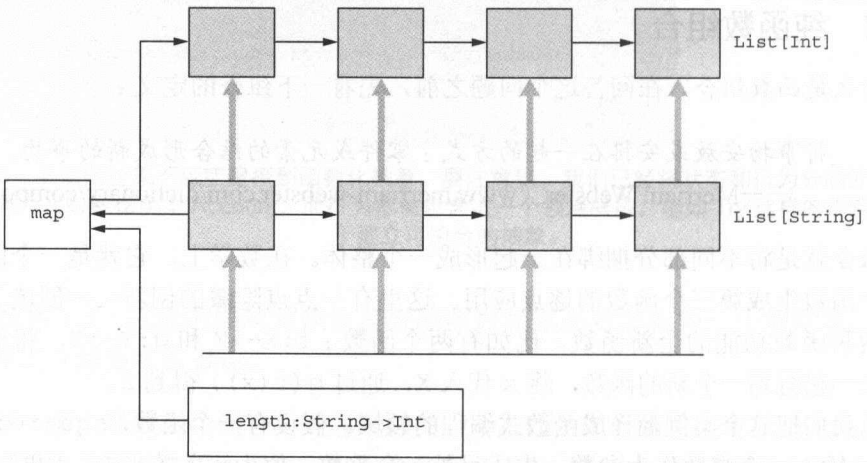


图 1.7 map 是一个将其他函数作为输入的高阶函数。

函数 map 有两个参数：一个字符串的 list 以及另外一个函数 length:String->Int。map 在 list 上进行迭代，并对 list 的每个元素应用函数 length，它生成的结果是另外一个 list，每个元素的结果是 length 的运行结果，于是得到一个整数

1 read-eval-print-loop，即读取—计算—打印—循环。——译者注

list。这是一个如何用函数思想思考问题的精彩案例，在表 1.2 中列出了这个范式一些有意思的特性。像 map 这种高阶函数也被称为组合器（combinator）。

表 1.2 结合 map 函数，用函数化的方式思考。

map 函数特点	如何关联函数式编程
可以将一个函数作为参数。在例子中，函数 length 作为 map 的参数	函数是最好的抽象
map 是一个将其他函数作为输入的函数	map 是一个高阶函数
map 遍历字符串 list，但遍历过程由 API 用户抽象	通过函数式编程，告诉函数要做什么。它如何做由 AIP 用户来抽象。也可以将 map 用于其他类型的序列（不仅仅是 list），并由 map 的实现来处理迭代过程

测验时间 1.5 如果传递给 map 的函数碰巧更新了一个共享的可变状态，这时候会发生什么？是不是意味着对一个 list 遍历多次会导致不同的输出？

清单 1.7 中的代码使用了高阶函数，如 Scala 里的 map，来演示组合的不同方式。每个例子都遵守表 1.2 中所列的指导原则。

清单 1.7 函数组合与高阶函数

```
scala> val inc1 = (n: Int) => n + 1
inc1: Int => Int = <function1>

scala> val square = (n: Int) => n * n
square: Int => Int = <function1>

scala> (1 to 10) map inc1
res1: scala.collection.immutable.IndexedSeq[Int] =
➥ Vector(2, 3, 4, 5, 6, 7, 8, 9, 10, 11)

scala> (1 to 10) map square
res4: scala.collection.immutable.IndexedSeq[Int] =
➥ Vector(1, 4, 9, 16, 25, 36, 49, 64, 81, 100)

scala> val incNSquare = inc1 andThen square
incNSquare: Int => Int = <function1>

scala> incNSquare(4)
res6: Int = 25

scala> val squareNInc = inc1 compose square
squareNInc: Int => Int = <function1>

scala> squareNInc(4)
res8: Int = 17
```

函数：对一个整数加 1

函数：对输入整数进行平方

Map：基于集合的递增函数

Map：基于集合的平方函数

定义函数组合（加 1，然后平方）

定义组合的另一种方式（平方，然后加 1）

现在是时候用一些组合器来充实 Balance 领域模型了。在复杂领域建模中，会有很多自己的组合器。但那些来自标准库的组合器非常管用，经常会发现在建立自己组合器的过程中，最后还是会回去使用标准库的组合器。毕竟，它代表了我们一直追寻的组合性。

测验答案 1.5 我们必须坚持一点，在使用 map 或其他组合器时，不要违背纯粹原则。可以传递任何函数给 map，不管有没有副作用或者可变性。后面内容会讨论副作用。

现在试着在个人银行系统中实现一些领域行为。假设想要对交易（比如 debit 和 credit）增加一个查账功能，同时生成查询记录并把它们记录在某个地方。在以下例子里会略过一些细节——这对于理解概念来说并不重要。假设有如下两个函数：

- generateAuditLog: (Account, Amount) => Try[String]
- write: String => Unit

这两个函数可以作为一个编程模式的简单练习。但我们现在的想法是用函数组合以及高阶函数来达到同样的目的。清单 1-8 描述了具体实现。

清单 1.8 通过高阶函数组合

```
val generateAuditLog: (Account, Amount) => Try[String] = //..
val write: String => Unit

debit(source, amount)
  .flatMap(b => generateAuditLog(b, amount))
  .foreach(write)
```

在账户中记入借方，debit 返回 Try[Account]，即 flatMap 的 #B

如果 debit 通过，生成查账记录

如果记录生成成功，写入数据库

这个需求被定义成一个函数，它执行如下序列的操作：

1. 在一个账户中执行 debit。
2. 如果 debit 通过，生成一个查账记录；否则，结束。
3. 将记录写入存储。

需要用函数式编程提供的组合器来准确实现这个序列。在这个序列中的行为流与领域行为的建模必须反映同一个序列。体会现在所掌握的函数化思想和前面讨论的组合器，清单 1.8 用同样的逻辑提供了一个真实的画像。

以下几点是清单 1.8 中的行为流程。如果大家跟我一样喜欢用图形的方式观察这类交互，就可以看图 1.8。

1. 调用 debit 可能生成一个 Failure（并产生一个异常），如果成功则生成一个变更后的 Account。

2. 在失败的情况下，整个序列全部中断直接结束。这里没有明确的失败检查。所有这样的样板行为都被隐藏在 map 组合器的实现中。
3. 如果 debit 成功生成一个 Account，这个值就被注入 flatMap 并传递给函数 generateAuditLog。
4. generateAuditLog 也是一个纯函数，它通过 foreach 来生成一个字符串记录日志。如果日志记录生成失败，序列中断，操作结束。
5. foreach 是一个组合器，它用于处理其他行为操作。序列的最后阶段会把日志记录写进数据库或文件系统，这是一个非常必要的附带行为。可以使用 foreach 来实现它。

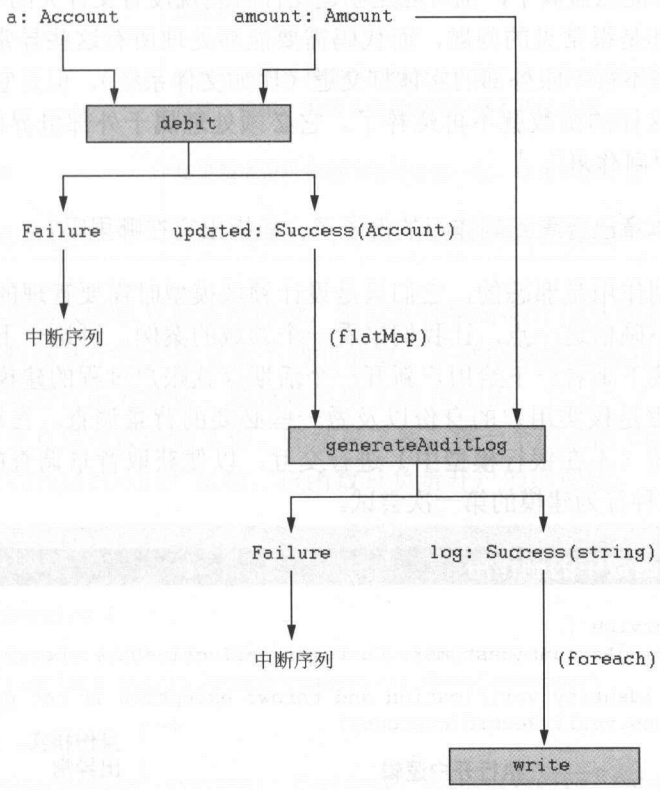


图 1.8 使用组合器通过函数组合改善领域行为。注意流程中的成功路径和失败路径。一次失败会立刻中断序列。

本节最主要的内容是，体会通过组合器函数组合是如何改善领域行为的。方法就是通过将小型的组合器组装成更大的行为以及函数化思想。在下一节中，将学习函数化思想如何改变代码，使我们可以推导它们，就如同数学里的函数一样。

1.4 管理副作用

到目前为止，已经讨论了很多纯函数的属性，这些都使得它们具备良好的组合性，也使得我们能够为领域设计完美的抽象。如果现实是如此简单，为领域模型所写的函数都如此纯粹，那么整个软件开发行业就不会看到那么多的失败项目了。

考虑一个简单的函数 square，输入一个整数并输出平方值。现在游戏规则变了，与之前那个不同，在输出整数平方值的同时，还要额外将结果写入文件系统的一个特定文件。在试着将输出结果写入文件时如果得到一个 I/O 异常，这时会发生什么？可能磁盘满了，也可能是创建文件时发现没有文件夹的写权限。

所有这些都是很常见的问题，而代码需要能够处理所有这些异常——这也就意味着函数现在不得不跟外部的实体打交道（比如文件系统），但是它们并不是输入的一部分。这样的函数就不再纯粹了。它必须处理属于外部世界的外部作用，我们也称之为“副作用”。[1]

测验时间 1.6 本章已经看到副作用的例子了，能指出它在哪里吗？

并不是说副作用是邪恶的，它们只是设计领域模型时需要管理的基本组件之一。如果还是不确信这一点，让我们来看一个领域的案例，讨论一下该如何处理这个副作用。接下来看一下给用户新开一个活期存款账户过程的建模场景。开户其中的一个步骤是核实用户的身份以及做一些必要的背景调查。在这个操作中，必须和其他系统（不在银行模型中）进行交互，以便获取背景调查的正确结果。清单 1.9 是对这种行为建模的第一次尝试。

清单 1.9 函数中的副作用

```
trait AccountService {
  def openCheckingAccount(customer: Customer, effectiveDate: Date) = {

    // does an identity verification and throws exception if not passed
    Verifications.verifyRecord(customer)          <-- 身份核实，如未通过则抛出异常

    //..                                          <-- 执行开户逻辑

    Account(accountNo, openingDate, customer.name, customer.address, ..)
  }
  //.. other service methods
}
```

openCheckingAccount 做的第一件事是调用 verifyRecord 来核实用户的身份。这个调用是和外部世界的交互，有可能会需要通过一个外部网关调用一个 Web 服务来做这个校验。这个调用可以失败，但代码却不得不处理与外部系统

1 在这里，副作用并不仅仅指负面作用，更多的是指超出函数本身的额外作用。——译者注

失效相关的异常。并且，这跟给用户提供一个合法的新开账户的核心领域逻辑没有一丁点儿关系！

这种情况没有办法回避，在探索领域模型实现的其他使用场景时，还会看到很多这样的情况。处理这个问题的最好方法是尽可能将副作用与纯领域逻辑剥离开。但在此之前，让我们整理一下把副作用和纯领域逻辑混合在同一个函数里的坏处，参见表 1.3。

表 1.3 为什么把领域逻辑和副作用混在一起是糟糕的。

混合副作用与领域逻辑	为什么是糟糕的
领域逻辑与副作用纠缠不清	违背了关系的隔离。领域逻辑与副作用彼此正交——纠缠在一起违背了基本的软件工程原则
难以开展单元测试	如果领域逻辑和副作用纠缠在一起，单元测试就会变得很困难。不得不依赖于模拟，但这又会在源码中引入其他问题
难以推导领域逻辑	如果领域逻辑和副作用纠缠在一起，就难以推导它
副作用会妨碍代码的模块化	代码中会存在一个孤岛，不能被其他函数所组合

我们需要重构代码以规避这种纠缠，也需要确保将副作用从领域逻辑中解耦掉。接下来的例子将这两个操作拆分成不同的函数，并确保领域逻辑只保留纯函数，这样就可以独立地进行单元测试。这就是清单 1.10 所做的。它引入了一个新的函数来承担与外系统交互以及校验的工作。在结束之后，将一个成功的 Customer 实例传给 openCheckingAccount 函数，该函数只负责开户的纯逻辑。

清单 1.10 解耦副作用

```
trait AccountService {
  def verifyCustomer(customer: Customer): Option[Customer] = {
    if (Verifications.verifyRecord(customer)) Some(customer)
    else None
  }

  def openCheckingAccount(customer: Customer, effectiveDate: Date) = {

    //..                    <------ 开户逻辑

    Account(accountNo, openingDate, customer.name, customer.address, ..)
  }
  object AccountService extends AccountService
}
```

管理副作用是一个非常重要的内容，它可以在组合领域模型与非组合领域模型之间产生巨大的差异。一个非组合模型经常要承受来自样本文件以及整合代码的各种折磨。这对维护来说就是一个噩梦——难以管理和扩展。但把副作用和纯逻辑剥离开，代码就会有更好的组合能力。在例子中，openCheckingAccount 再次成为一个纯逻辑。清单 1.11 显示了如何将 verifyCustomer 和 openCheckingAccount 结合在一起，以及处理副作用的代码部分如何应对可能发生的失败。

清单 1.11 组合身份验证与开户

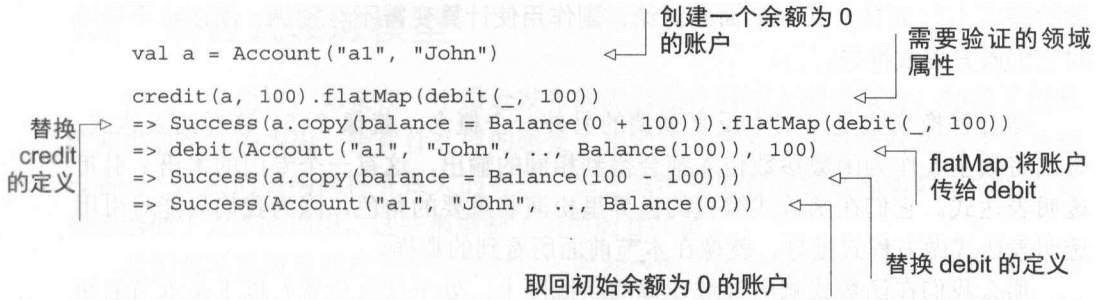
```
import AccountService._

val cust = getCustomer(..)
verifyCustomer(cust).map(c => openCheckingAccount(c, date))
  .getOrElse(throw new Exception(
    "Verification failed for customer"))
```

测验答案 1.6 在清单 1.4 中已经看到了副作用的例子，领域行为 debit 和 credit 需要更新一个共享可变状态。注意，副作用依赖于外部系统的任何事物，也许会涉及文件系统，或一个全局可变状态。

1.5 纯模型元素的优点

如果对纯函数进行了组合，领域模型（或至少部分领域包含纯函数）在一定程度上就会展现一些数学属性，对吗？至少这是我在鼓励大家用纯函数建模时的想法。在本节中，将试着搞明白是否能够证明模型中的一些属性并验证它的正确性，就像数学中那样。这也被称为*方程式推导*，在后面的章节中会看到很多例子。

回顾清单 1.6 中的定义，其中将 debit 和 credit 实现为纯函数。通过这种模型，可以写断言来验证实现。这也被称为模型的*属性*，下面的代码片段提供了模型的*正确性校验*。从一个表达式开始，连续对一个账户执行相同数目的 credit 和 debit。在执行这个表达式之前，会提供一个原始的账户余额。为了校验这个属性，将每一步执行都替换成推导（就像在解决一个数学公式一样），并追踪获取结果。



推导中获得了什么呢？我们证明了一个明显的论点，就是对一个账户执行一个数目为 x 的 credit，并紧接着执行一个相同数目的 debit，不会改变账户的余额。是不是非常明显？

测验时间 1.7 判断对错：方程式推导和副作用不会同时发生。

当我们用数学的方式对待函数调用时，上述论点就非常明显。将函数调用替换成它的实现并期望不管重复多少次都能得到相同的结果。参考清单 1.12 中的例子[1]。在这个例子中，跟踪一个函数调用 f(5)，重复用函数体以及相应的参数替换调用。这和心算函数的方式非常相似。在函数式编程的理论里，这被称为赋值的替换模式。

清单 1.12 赋值替换模式

```
def f(a: Int) = sum_of_squares(a + 1, a * 2)

def sum_of_squares(a: Int, b: Int) = square(a) + square(b)

def square(a: Int) = a * a
```

- f(5)
- sum_of_squares(5 + 1, 5 * 2)
- square(6) + square(10)
- 6 * 6 + 10 * 10
- 36 + 100
- 136

你看，使用替换模式，用参数值 5 调用函数 f，每次都能得到相同的结果。但要清楚的是，只有函数为纯函数，同时没有不可控的副作用时，替换模式才能正常工作。如果平方函数除了返回一个值还要写入文件，那就包含了副作用，因为有些调用的文件写操作可能会失败，函数将生成一个异常。那么替换模式每次调用都生成相同值的保证就落空了。这也是我们青睐纯函数的另一个理由。

1 这个例子是受了*Structure and Interpretation of Computer Programs*（Harold Abelson）的启发。

测验答案 1.7 对的。正如前面的讨论，副作用使计算变得不可预测，所以也不可以对它们做方程式推导。

结合替换模式是本书中反复强调的另外一个概念。就像 `f(5)` 这样的表达式，每次将数字 5 作为函数参数传入都会得到相同的输出，这有一个专门的术语：引用透明表达式。它们在函数式编程的世界里扮演着重要的角色，因为我们只能对引用透明表达式做方程式推导，就像在本节前面所看到的那样。

那么我们在函数式响应领域建模的路线图上，处于什么位置？以下是本节的快速汇总：

- 引用透明表达式是纯粹的。
- 引用透明表达式是替换模式的前提。
- 用替换模式协助方程式推导。

在图 1.9 中，总结了函数式编程的三大支柱。

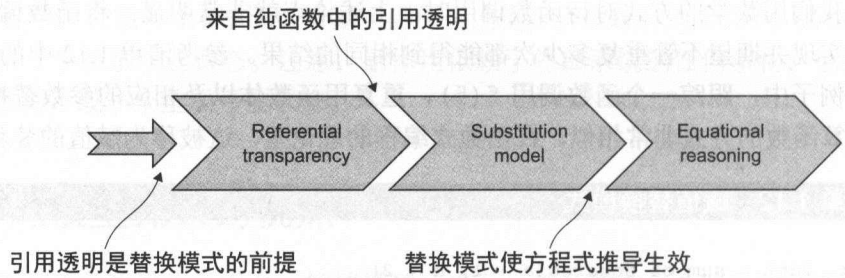


图 1.9 引用透明在函数式编程中扮演重要角色。

现在已经讨论了函数式编程的基本原则，这都有助于建立一个更好的领域模型——更好意味着可以使模型：

- 利用函数组合的力量，用小函数组装成一个大函数，获得更好的组合性。
- 纯粹，领域模型的很多部分都由引用透明表达式组成，它们具备我们讨论的很多良好的特性。
- 通过方程式推导，可以很容易地推导和验证领域行为。

下一节将聚焦在另一个方面：让模型更具有响应性。用户不喜欢太长的等待时间，不管是查询账户余额，还是打电话到银行开一个活期账户。软件需要在一个合理的时间内响应所有的用户行为，我们经常称之为*延迟*。同时，它也是使模型在一个边界延迟下工作的*响应式*属性。

1.6 响应式领域模型

作为一个用户，如果在一个最近发布的酒店预订网站上查询报价，却花了很长时间来等待它的响应，会有什么样的感受？相信我，不要总是责怪网络或者是基础设备——应用的架构同样有很大的责任。也许领域模型对底层资源如数据库或消息服务器做了太多的调用，这些都遏制了应用的吞吐量。

我们都希望应用能在一个可接受时间周期内响应用户的请求。这个时间被称为延迟，它通常被定义为从发出请求到从服务器获得响应之间所流逝的时间周期。如果能确保将这个延迟限制在一个可接受范围内，那么就达到了响应性目标。保持良好的响应能力是响应式模型的基本判断标准。但如何应对失效呢？一个夹杂着不清楚原因的失败的系统是没有响应能力可言的。解决这个问题的关键是围绕失败来设计系统，在 1.6.2 节中会看到更多这方面的内容。

一个响应式模型的基本特征是什么？在表 1.4 中汇总了其主要标准。

表 1.4 保证模型的响应：4 个属性。

map 函数特点	如何关联函数式编程
积极响应用户的交互	否则没人会使用我们的应用
弹性	意味着要积极响应失败情况。如果系统在面对失败时陷入未知状态，那就说明没有成功建立一个稳定的模型。必须要重启部分应用模型，或者给用户一个恰当的反馈，指导用户下一步操作
伸缩性	意味着在不同负载的情况下保持良好的响应能力。系统可以承受负载的波动，就算面对高负载，也要能够控制住延迟水平
消息驱动	为确保弹性和伸缩性，系统必须保持松耦合，并通过使用异步消息将阻塞最小化

1.6.1 响应式模型的 3+1 视图

如果仔细看了表 1.4 中的 4 个属性，就会注意到实现响应式模型的关键在于使其具备积极的响应能力。而另外 3 个属性恰恰是响应能力不同的表现形式。这也正是观察响应式领域模型的另外一种方式——我们称之为模型的 3+1 视图，如图 1.10 所示。

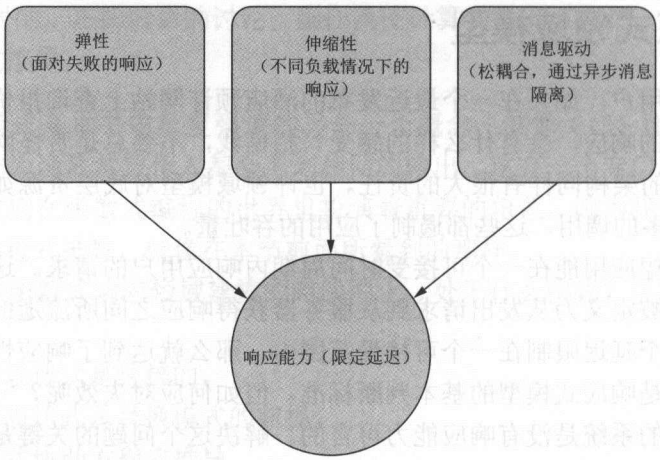


图 1.10 响应式领域模型的 3+1 视图：模型的响应能力可以通过 3 个方面来获得，对失败的响应、对负载的响应以及异步消息。

测验时间 1.8 我们最喜欢的网上书店在一年的大部分时间里都能正常工作。但在圣诞节和其他假期中，它的响应速度大幅下降。它违背了哪一条响应标准？

模型的响应能力基于 3 个方面：弹性、伸缩性以及并行性。每个方面都有不同的实现策略，本书后面会分别讨论。

1.6.2 揭穿“我的模型不能失败”的神话

那些天真的建模工程师通常有一个错误的观念，那就是他们的模型不能失败——他们认为自己已经处理了所有可能发生的异常。但现实往往是相反的，不管我们多么坚定地相信自己已经管理了模型所有的异常，失败也总会发生。而且模型的规模越大，组件失败的概率也就越大。磁盘错误、内存错误、网络组件错误、其他基础设施错误——总之，失败的发生总是超出了我们的控制能力。

> 针对失败进行设计。这是开发由很多互相协作的组件组合而成的大型服务时的核心思想。这些组件都会失败，而且还会失败得非常频繁。这些组件不会总是互相协作，也不会孤立地失败。一旦服务规模超过 10000 台服务器以及 50000 块磁盘，失败就会在一天之中发生多次。
>
> ——James Hamilton
>
> *On Designing and Deploying Internet-Scale Services*
>
> www.usenix.org/legacy/events/lisa07/tech/full_papers/hamilton/hamilton_html/

事实上，响应式模型最主要的一个方面就是围绕失败来做设计，并提升模型的综合弹性。表 1.4 中谈到对失败的响应，意味着模型不仅仅必须具备处理来自应用内部失败的能力，同样还要能够处理来自外部的失败。这并不意味着会有各种异常处理逻辑会污染领域模型。基本的概念是，接受失败是不可避免的，而且它们发生在系统的各个部分，所以要明确实现处理失败的相应策略。

我们来看一个例子：一个用户要求计算她在银行中不同账户的投资组合。在计算过程中，某个步骤失败了，可能因为保存某个账户余额的服务器不可达。在应用中如何处理这个失败？看起来至少有两种解决方式：

- 试着在计算投资组合的应用代码中包含异常处理逻辑。但这对所有 API 来说，导致的结果就是要访问后端的服务器。然后再想象一下，在所有存在同样异常的地方复制一遍同样的处理代码。这个结果是软件工程的一个灾难——这也被称为未能成功处理*关注点分离*。我们在领域逻辑里合并了异常处理代码，但最后却使得后面的代码污染了前面的。很显然这并不合适。
- 用一个独立模块来处理失败。所有失败都委托给这个模块，由它基于用户定义的策略来负责处理所有失败。这种方式能保持领域模型的纯洁，并把失败处理从业务逻辑中解耦出来。

图 1.11 总结了这两种方式。（见下页）

于是我们一定会想，如果所有失败都通过一个模块来处理，那对整个模型来说它会不会成为伸缩性的瓶颈呢？如何才能保证失败处理和其他处理领域逻辑的模块一样可扩展？

测验答案 1.8 网站在面对负载增长时难以保持响应能力——也就是说它不能随着流量的增长而扩展。因此，它违背了伸缩性准则。

解决方案还是在失败处理模块的设计本身。并不是说在整个应用中必须只有“一个”单独的模块来做失败处理，而是说我们需要集中地处理。但需要多少个模块，取决于整个应用的模块化情况。在前面的例子中，可能需要一个模块来处理所有计算失败的投资组合。

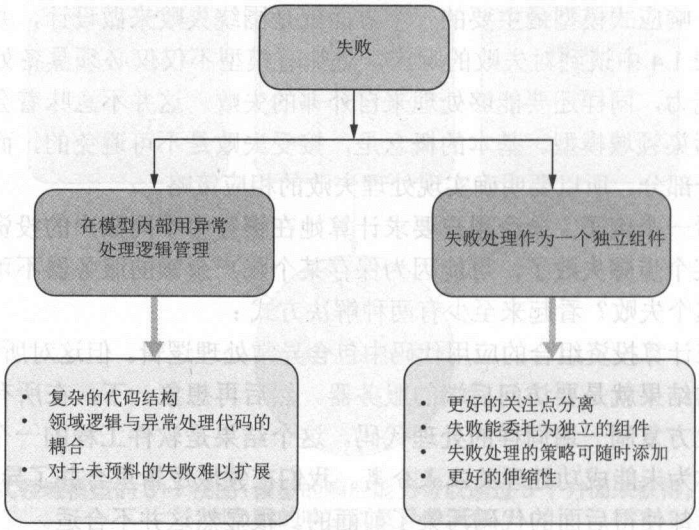


图 1.11 领域模型中处理失败的策略。针对失败设计独立的处理模块通常（甚至绝大多数情况）是更好的方式。

1.6.3 伸缩性与消息驱动

当谈到伸缩性时，就意味着系统必须能够适应不同的负载。也就是说，在负载增长时系统可以扩张，在负载下降时可以收缩。在系统安静的时期，随着负载下降系统能相应收缩是非常重要的，因为这可以节约资源与确保操作的成功。

当系统的负载上升时，比如遇到了节假日，就会看到延迟的瞬间峰值超出了我们向客户承诺的服务水平协议（SLA）。保持伸缩性也就意味着系统可以适应不同的延迟水平。

有一个方法可以使系统拥有伸缩性，那就是减少模型组件之间的耦合。宽松连接的架构，使用异步消息作为沟通的途径。这也恰恰就是响应式模型所鼓励的——非阻塞通信，组件的交互通过不可变的消息，不使用任何共享的可变状态。当组件用异步消息进行交互时，就说明我们已经有了一定的隔离水平，因为已经可以忽略位置、并发模型以及编程语言本身。

本书聚焦在使用 actor 模型的消息系统。actor 模型提供了一个非常高水平的并发结构，这有助于用松散组合的模块（也可以是边界上下文）来组织模型，它们之间的交互都通过异步消息。接下来将讨论一个良好组织的 actor 系统是如何为模型提供弹性的，以及如何通过扩展或收缩来应对背后的压力，并最终为系统创造全面的响应能力。

在说到消息驱动时，本书有意让其显得更宽泛一点。事件同样可以被认为是消

息，它囊括了一个领域概念。当本书提到一个 *debit* 消息，它其实是一个事件，针对特定账户的 debit 操作。而同时，*debit* 是一个领域概念。在下一节中将会讲到领域事件如何在响应式模型中构建一个组合行为的头等构架。

1.7 事件驱动编程

在对事件是什么有了一点概念之后，让我们从个人银行领域开始。从一个所有函数均为同步调用的模型开始，所有的阻塞和执行都是完全串行的。你会学到这种模型的缺陷，并尝试通过引入一个事件驱动架构来提升其响应能力。

考虑这样一个功能，作为一个用户，如果向银行索要所有资产的投资报告，那么这里有一些典型的项目需要抓取、计算并汇总，最终生成报告：[1]

- General currency holdings
- Equity holdings
- Debt holdings
- Loan information
- Retirement fund valuation

下面是这些项目的第一种实现方式，它的结构如清单 1.13 所示。

清单 1.13 投资组合报告

```
val curr: Balance = getCurrencyBalance(..)
val eq: Balance = getEquityBalance(..)
val debt: Balance = getDebtBalance(..)
val loan: Balance = getLoanInformation(..)
val retire: Balance = getRetirementFundBalance(..)
val portfolio = generatePortfolio(curr, eq, debt, loan, retire)
```

从中可以看到，它是一个串行的代码块，执行其中任意一个，都会阻塞主线程的执行。只有当序列中一个函数执行完成，并在执行的主线程中生成合法的结果之后，下一个函数才能接着执行。其导致的后果是，计算的总延迟就是所有独立函数延迟的和，如图 1.12 所示。

1 以下均为资产项目，中文意思是货币资产、股权资产、债务、贷款、退休金，考虑到与下文中的函数名相呼应，故此处保留英文。——译者注

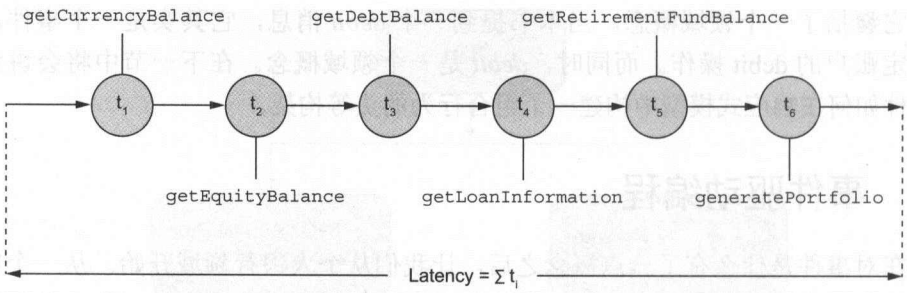


图 1.12 在串行执行中，总的计算延迟就是每个函数的延迟总和。没有任何并行，计算是严格的串行。

这种情况会严重伤害响应能力，因为有些函数可能会访问数据库或其他基础服务，这可能会需要等待很长时间才有响应。作为一个用户，一定不愿意一直瞪着计算机屏幕，等待后台基础服务在一长串计算的漫漫征途上艰苦跋涉。而我们马上就会看到，事件可以在一定程度上缓解这种糟糕的体验。

前面代码未能兑现承诺的另一个方面就是它的架构仅支持本地执行模型。[1]但当投资组合的不同组件需要通过网络访问一个计算机集群或者多个服务而不只是一台机器时，之前的代码就彻底傻眼了。而这也恰恰是基于事件的模型能成为解决方案的很重要的一个方面。

让我们重新梳理下前面的代码，然后换种方式来组织它，把处理分发给多个并行计算单元，让主线程承担协调者的角色（参见清单 1.14）。

清单 1.14 投资组合报告——事件驱动

```
val fcurr: Future[Balance] = getCurrencyBalance(..)
val feq: Future[Balance] = getEquityBalance(..)
val fdebt: Future[Balance] = getDebtBalance(..)
val floan: Future[Balance] = getLoanInformation(..)
val fretire: Future[Balance] = getRetirementFundBalance(..)

val portfolio: Future[Portfolio] =
  for {
    c <- fcurr
    e <- feq
    d <- fdebt
    l <- floan
    r <- fretire
  } yield generatePortfolio(c, e, d, l, r)
```

1 当执行模型如同现在这个场景是串行阻塞时，可以做远程调用的同时给用户呈现一个本地执行模型的界面。但是它永远无法扩展，并最终成为分布式计算模型的牺牲品。详情参见Arnon Rotem-Gal-Oz所著的*Fallacies of Distributed Computing Explained*（http://www.rgoarchitects.com/Files/fallacies.pdf）。

在这里，每个独立函数都不再承诺在将控制权还给主线程前返回 Balance，而会返回一个 Future，我们可以将它理解为该计算的某种占位符。Future 承诺在函数技术结束之后，会最终给我们一个 Balance。

不过因为几乎是瞬间行为，所以它不会阻塞主线程。主线程可以继续执行其他任务，而这个独立函数可以用其他执行线程来兑现它的承诺。每个独立函数在它们各自的线程中执行，而主线程仅仅作为协调者并对结果进行汇总。这也恰恰是调用 generatePortfolio 函数时所发生的情况——它收集所有结果，然后计算投资组合报告。

如果这是成本很高的操作，还可以将这个计算派发给一个 Future（如清单 1.14）。作为一名机智的读者，一定已经发现，在这种场景下，计算的总延迟就是所有涉及余额计算函数的延迟中最大的那一个，也就是 generatePortfolio 计算的延迟。[1] 但因为已经将整体计算函数委托给了一个 Future，因此主线程的执行就被空了出来，它可以去服务其他请求，而不用再苦苦等待这些函数的完成。这个时候模型在响应能力上已经有所提升！[2]

清单 1.14 中 Future 的计算已经从清单 1.13 的串行阻塞式的代码变为异步非阻塞的代码。当计算返回一个 Future，就相当于给调用线程发了一个事件，它说："当我完成之后给你一个可用的结果。"于是在计算完成之后，调用线程就获取一个事件，表明结果已经可用了。然后就可以通过之前注册的回调来获取结果。现在所看到的是事件驱动编程的一种形式，在这里事件被 Future 隐蔽地发送给了调用线程。

1.7.1 事件与命令

我非常确信，大家现在已经非常开心地把事件当作小型消息用于非阻塞编程模型，并且已经知道期货（future）如何与执行的主线程交互，然后并行分发给领域模型。当完全非传统的领域模型工作时，会遇到非常多这样的场景，凭借着事件可以使模型更具有响应性。

让我们想一个事件 Debit，从账户中借钱。现在账户余额发生了变化。如果在数据库中维护了余额，那么这个事件将触发更新，会修改余额。如果在内存中同时还维护了一份汇总的副本，就还要根据账户新的余额刷新下一内存。

---

1 第6章中会更详细地讨论为什么总的延迟时间是"最大的那个"而不是"所有的和"。

2 我们所讨论的关于通过使用事件以及异步编程模型所带来的性能提升，并不是异步非阻塞计算带来的全部真相。在阻塞情况下，提升CPU操作通常也能带来性能提升，它们可以从缓存一致性和更少的调度开销中获利。

一旦账户发生借款，手机就会收到一条来自银行的信息，提示账户发生一笔数目为 x 的借款，形式为现金提款。这个消息被命名为 DebitOccurred。事实上，这也是一个事件。

大家能看出模型中这两种交互之间有什么不同吗？表 1.5 列出了它们的差异。

表 1.5 两种事件的解释。因为性质不同，有时候 Debit 被称为命令，而 DebitOccurred 被称为事件。

Debit	DebitOccurred
对系统全局状态产生作用，针对汇总做写操作，在当前场景下改变账户的余额	只是发送一个通知给感兴趣的订阅者——在这个案例中，就是账户的所有者
系统内对象在系统中产生结果之前发送消息	系统产生结果之后发送消息
作为一个可变消息，通常被一个单独的系统处理者处理	可以被多个部分处理，对消息的响应也各不相同
如果违反某些约束就会导致失败	不会失败，因为相关的结果已经在系统中产生了

我们称 Debit 是一个命令，而 DebitOccurred 是一个事件。两者都是模型生成并处理的消息，但它们在语义上有微妙的不同。留意这些区别，在讨论领域模型架构时，会使用完全不同的方式来应对命令和事件。

1.7.2 领域事件

事件驱动编程模型使事件成为一个非常重要的架构元素。事件会触发领域逻辑，并且在领域模型内参与各种交互。用更通用的术语来说，事件就是通知的一种形式。在前面的章节中，有两种类型的通知，在语义上两者有轻微的不同：命令和事件。我们经常用术语“事件”来统称这两种类型的通知，只有一种例外情况，那就是命令需要用不同方式来处理。

在前面一节中，有一个 Debit 事件的例子，它触发了从银行账户中借款的行为，还有一个 DebitOccurred 事件，它会通知那些感兴趣的部分某个账户发生了借款。根据这些事件在领域模型内执行的行为会给它们不同的命名，这些事件说的也是领域的语言。因此它们也被称为领域事件。

这些事件的一个重要特征是它们是不可变的。这是一种直觉，因为我们看得出来这些事件是那些在系统中已经发生的事情。所以，在过去已经发生的事情怎么可能改变呢？

作为银行的用户，Bob 在 1989 年 3 月 1 日注册了地址 A。在 2010 年 6 月 6 日，他搬去了地址 B。针对这种情况，通常的处理方式是在用户实体中更新 Bob 的地址。

但讨论事件时，如何才能发出一个更新声明并且在用户记录中更新 Bob 的地址？他现在居住在地址 B 的情况并不能颠覆他过去住在地址 A 的事实。大家也许会说，在数据库服务器维护的日志中依然能够指明 Bob 在过去的某段时间居住在地址 A。但是，不管如何，数据库日志不是我们模型的组成部分，而 Bob 在过去某个时间住在地址 A 的事实却是我们设计的领域模型语义中非常重要的组成部分。让我们看一下在领域模型中所发生的事件序列，表 1.6 显示了 Bob 的时间线变更历史。

表 1.6 Bob 时间线里的事件序列。

时　间	动　作	事　件
T0	Bob 新开一个账户，地址为 A	触发一个事件，`AddressChanged(Bob,_,A,T0)`
T1	Bob 的地址变更为 B	触发一个事件，`AddressChanged(Bob,A,B,T1)`

在这里讨论一下与基于标准关系型数据库模型的另外一种建模方式。我们现在讨论的事件是不可以改变的，而且它们还被应用到领域模型中来达成当前的快照。在前面的例子中，可以应用 Bob 在银行中记录的最后的事件来或获取他当前的地址，同时也不会丢失他过去有另外一个不同地址的信息。这就意味着我们不仅拥有模型的当前快照，还拥有生成这个快照的完整历史日志。图 1.13 很清楚地描述了这个特征。

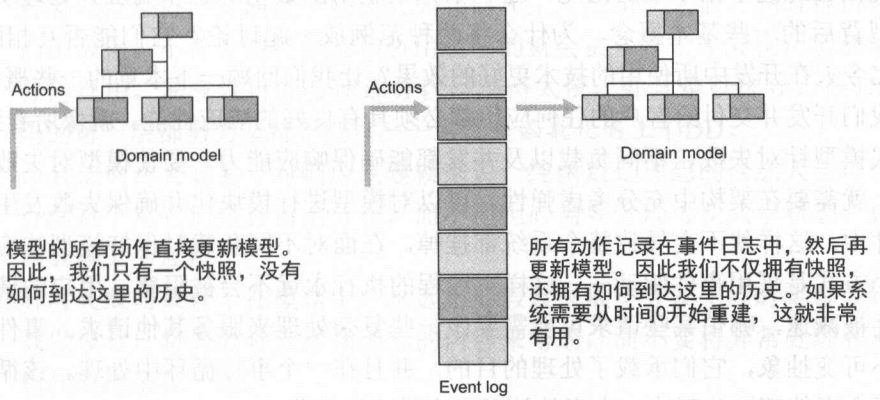


图 1.13 比较两种模型风格。左边的模型接受所有动作并将它们更新到当前快照中。因此丢掉了变更的历史。而右边的模型将所有事件存储在一个日志中。这样就保留了所有历史，于是也就能够从任何一个节点重建整个模型。

前面讲过，领域事件在搭建响应能力良好的模型架构时有很重要的作用。也简要介绍了它们是什么以及它们如何帮助我们建立模型，同时保存完整的进化史。这种领域模型被称为自追踪模型，因为领域时间日志使我们可以在任何时间节点追踪模型。

所以领域事件在响应式领域模型中是一个非常重要的参与者，在建立自己的模型时必须给予它们足够的重视。Jonas Boner 在一篇关于事件驱动架构的文章中对领域事件是什么做了很好的总结。[1] 他认为领域事件是：

- 唯一定义的一个类型：在模型中，针对每个事件都有一个相应的类型。
- 自包含作为一个行为：每个领域事件都包含系统中刚发生变化的所有相关信息。
- 用户可见：模型中的下游组件为了进一步的行为可以消费事件。
- 时间相关：这可能是领域事件最重要的特性。一个时间的单调性被建立在事件流中。

如果领域模型由从时间 t_0 开始的所有领域事件构成，那么完整模型就是下面的数学等式：

$$M(t_n) = \Sigma(\text{all events from time } t_0 \text{ till } t_n)$$

$M(t_n)$ 就是模型的状态，也被描述为从时间 t_0 开始的所有事件的和。

在后面的章节中，会看到这个等式如何映射函数式编程中一个同样的概念。

1.8 函数式遇上响应式

现在已经到了第 1 章的结尾，这一章介绍了用函数化思想和响应式处理实现领域模型背后的一些基本概念。为什么将两种范例放一起讨论？它们能否互相配合，产生比今天在开发中所使用的技术更好的效果？让我们回顾一下本章的一些概念。

我们开发并交付给客户的任何应用都必须具有良好的响应性能。就像所看到的，响应式模型针对失败、不同负载以及并发都能确保响应能力。要使模型对失败保持灵敏，就需要在架构中充分考虑弹性。可以对模型进行模块化并确保失败发生在一个组件中，这样就不会导致整个系统都挂掉。在面对不同负载时保持模型响应能力的一个方式是采用事件驱动——这样主线程的执行永远不会被阻塞。用户的请求永远不会被减速，哪怕某些请求可能需要做一些复杂处理来服务其他请求。事件是小型的不可变抽象，它们承载了处理的目的，并且在一个事件循环中处理。该循环获取的每个事件都被分配给一个事件处理者来做实际工作。

响应式模型可以帮助代码良好地模块化，这样不同的事件处理者可以独立运行，并且用执行领域行为的方式工作。只有当事件之间没有或只有极少的共享状态时，才可以独立地运行处理。函数式编程从一开始就鼓励这种实践。用纯函数做设计，

1 参见：Jonas Bonér、Patrik Nordwall、Andreas Källbarg所著的*Building Loosely Coupled and Scalable Systems Using Event-Driven Architecture*（www.slideshare.net/jboner/event-drivenarchitecture-6097206）。

从纯逻辑中分离副作用，这也是函数思想的两个最基本的信条。纯的引用透明模块将扮演事件处理器，它们可以并发执行领域逻辑，使模型保持响应性和弹性。纯逻辑会保持伸缩而副作用不会，这也是把函数化思想和响应式建模结合之后所能获益良多的地方。

1.9 总结

本章开启了用函数式和响应式领域建模的旅程。我们已经学到了一些这两种方式的优点。函数式编程是基于函数的组合：通过组合函数作为语言的头等产物来建立抽象，利用响应原则来保证应用的响应能力。以下为本章总结的几个结论。

- *在模型内避免共享可变的状态*：共享可变状态非常难以管理，在语义上也会导致不确定性，进而导致并发的不可控。
- *引用透明*：函数式编程使我们具备设计引用透明（纯）模型组件的能力。当模型行为绝大部分都由纯函数组成时，我们就具备了组合的能力，能够通过组合，用小函数来建立更大的函数。
- *有机增长*：通过函数式设计和思考，模型将有机地增长。因为它是纯粹的，模型可以用数学的方式来对待处理，并且推导它。
- *聚焦在核心领域*：在用领域驱动设计的原则建立模型时，我们拥有实体、值对象以及服务，并通过一些模式如仓储和工厂来组织它们。同时也可以使它们函数化。不符合纯粹原则和引用透明的就会被当作一个异常，但必须有充分的理由。易变性可能会使部分代码跑得太快，并且难以推导。在 DDD 代码的每一层都努力坚持不变性——这就是*函数化遇上 DDD*。
- *函数使响应更容易*：对响应式建模来说，纯函数是理想的候选人。可以在并行配置下对它们进行自由分布，而不用担心需要管理可变共享状态。这就是*函数遇上响应式*。
- *针对失败的设计*：在模型中，永远不要假设失败不会发生。时刻都要针对失败做设计，将失败管理作为独立的内容来考虑，而不要将异常处理和业务逻辑代码混杂在一起。
- *用基于事件的建模来补充函数式模型*：基于事件的编程从模型中勾画出目的或结果“是什么”。这同样也是函数式编程所鼓励的。事件是小型的消息，它指出我们想做的“是什么”，而事件处理者描述的是“如何做”。难怪函数式编程和事件驱动编程能一起愉快地玩耍。

2 Scala与函数式领域模型

本章包括

- 理解为什么 Scala 是一个更好的领域建模语言
- 理解使用静态类型语言进行领域建模的好处
- 结合 Scala 的 OO 和 FP 的力量来实现模块化的纯模型

现在已经比较熟悉函数响应式领域建模的基本概念了，那么问题来了，如何实现一个这样的模型呢？在考虑实现时，必须考虑使用什么语言来实现。很多语言都有足够的能力来实现可表达的领域模型。本书使用 Scala，一种静态类型的面向对象的函数式语言，它运行在 Java 虚拟机（JVM）上，同时能和 Java 极好地协同工作。

大家也许会想，为什么是 Scala？对于领域模型的实现，Scala 提供了什么样的特性，使其具有足够的吸引力？在本章中将学到其中一些特性，同时看到在建立模型的特定方面它们是如何提供帮助的。Scala 同时具备了面向对象（OO）以及函数的能力，这在实现和组织领域模型时成为了一个强有力的组合。我们将用 Scala 的函数能力来实现不可变数据和领域行为，用它的 OO 能力来模块化领域模型。

图 2.1 描绘了本章的学习节奏，以及如何提升建立函数响应式领域模型方面的知识。

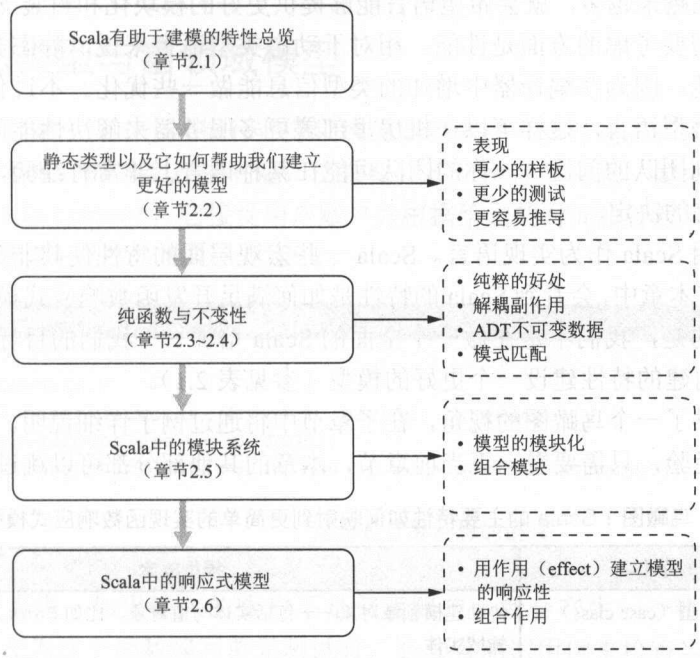


图 2.1 本章学习过程。

2.1 为什么是Scala

Scala 的特性使其成为适合于实现函数响应式领域模型的语言，在开始了解这些特性之前，先简要回顾一下第 1 章中学到的模型应该具备的几项特质。建模最重要的两个特质是函数式和响应式。一个模型是函数式的，也就意味着模型行为是用函数的方式实现的，同时副作用和纯业务逻辑是清晰分开的。这有助于模型通过函数组合的方式逐步演进，同时确保可以通过方程式推导的方式推导模型。一个模型是响应式的，意味着该模型在不同形态下都有良好的响应能力——不管是失败还是变化的负载都能快速响应——同时模型的整体架构能够确保用户不会成为巨大延迟的受害者。

要建设一个设计良好同时又对用户快速响应的系统，需要在开始设计架构时就做出正确的决定。这可能不是最重要的决定，但选择一个包含合适特性的实现语言是关系到系统整体稳定性的基本因素。

在选择所使用的语言时，需要考虑很多方面。可以选择一个动态类型的语言，

它拥有快速应用的开发平台，同时还有一个活跃的社区。这样就可以无所畏惧地快速完成模型的第一个版本。但从长远来看，模型的日子不会太好过——随着在模型中添加的特性越来越多，就会希望语言能够提供更好的模块化和封装支持。

另一个需要考虑的方面是性能。相对于动态类型语言来说，静态类型语言能提供更好的性能，因为在编译器中增加的类型信息能做一些优化。不过你可能依然想要使用动态类型语言，这样可以在机房里部署更多服务器来解决性能问题。但是这可能还牵涉到团队的问题——你的团队可能在某种语言上非常有经验，这也会影响选择实现语言的决定。

本书使用 Scala 作为实现语言。Scala 一些宏观层面的特性使其非常适合做服务端的开发。在本章中，会看到 Scala 的特性是如何满足开发函数响应式模型的需求的。不过要注意的是，我们不是在做一个全面的 Scala 培训——我们的目标是展现 Scala 如何通过它内建的特性建设一个更好的模型（参见表 2.1）。

本章提供了一个鸟瞰图的视角，在子章节中将通过例子详细说明。如果在 Scala 上有丰富的经验，只需要看一下当前章节，本章的其他部分都可以跳过。

表 2.1 鸟瞰图：Scala 的主要特性如何映射到更简单的实现函数响应式模型元素。

Scala 特性	模型概念
内建代数数据类型（case class）支持不变性	协助建模领域对象——包括实体与值对象，比如 `Bank`、`Account` 等领域实体
纯函数	协助建模领域行为，比如在个人银行系统中实现 `debit`、`credit` 等业务逻辑
函数组合与高阶函数	通过将小的行为组合成为大的行为，可以组合 `debit` 和 `credit` 来实现两个账户之间的转账逻辑
具有类型推断的高级静态类型系统	在类型本身里封装一些约束和业务逻辑，使模型更加健壮。类型推断使代码更加简洁，因为编译器能够从表达式推断类型
trait（特质）和对象的组合	Scala 的 trait 有助于模块化。可以通过将多个 trait 组合成对象来组织模型，实现不同的函数功能。trait 通过类型进行参数化，也就允许插入一些行为以满足特定的业务规则
支持泛型	帮助建立泛型的抽象，后期再具体实例化。比如，可以为通用用户类型 `C` 定义一个领域服务 `PortfolioService[C]` 模拟服务的共同工作流。可以在模型中为各种类型的用户针对不同部分进行重新定义
支持并发模型，如在 Java 并发模型之上建立的计算 actor 模型	Scala 支持并发的抽象，诸如 actor 和 future，这有助于建立响应式非阻塞性元素模型，而不需要再写线程、锁相关的底层代码

在后面的章节中，结合个人银行业务系统的案例，将详细了解表 2.1 中列出的特性。

2.2 静态类型与富领域模型

让我们结合第 1 章中个人银行业务领域的案例来进一步讨论。在银行里，只有用户账户是储蓄账户时才会产生利息，支票账户不产生任何利息。如果实现一个函数 calculateInterest，它接受用户账户并根据给定的时间段计算利息，应该如何建模呢？清单 2.1 是我们的第一次尝试。

清单 2.1 为一个用户账户计算利息

```
case class Account(..)
def calculateInterest(account: Account, period: DateRange) = {
  if (!(account.accountType == SAVINGS))
    Failure(new Exception(s"$account has to be a savings account"))

  Success(..)                          这里是计算利息的逻辑
}
```

对每一次需求，如果输入的账户不是储蓄账户，就会产生一个错误，否则就会计算利息。在为这个函数编写单元测试时，必须有测试用例来检查当遇到能被接受的特定类型的账户时是否计算利息。我们预期这个函数在传入一个非储蓄类型的 Account 时将产生一个错误，同时测试也必须能反映出这种业务校验。

让我们在这个实现中做个小小的修改：把函数改为多态的，不再传入具体的 Account 类型。所谓多态，就是指不用再像清单 2.1 中的 calculateInterest 那样输入一个具体的数据类型 Account，只须根据函数获取的账户类型来参数化，这使得函数能够处理的账户类型，实现了函数的多态化。可以使用一个不受限制的泛型类型 A，这也意味着可以传任意类型的账户给函数。在清单 2.2 中，强制设定类型参数 A 作为 InterestBearingAccount 的子类型。在这样的定义下，就不能再输入任何超出该限制的账户给函数。通过利用类型系统的威力，我们已经将某些领域逻辑编码到了类型中。

清单 2.2 计算利息——多态版本

```
trait Account {
  def number: String
  def name: String
  //..
}
case class CheckingAccount(..) extends Account

trait InterestBearingAccount extends Account {
```

```
  def rateOfInterest: BigDecimal
}
case class SavingsAccount(..) extends InterestBearingAccount
case class MoneyMarketAccount(..) extends InterestBearingAccount

def calculateInterest[A <: InterestBearingAccount](account: A,
➥ period: DateRange) = {
                                        <-- 这里是计算利息的逻辑
}
```

Account 现在是一个多态的数据类型，函数 calculateInterest 也被称为在 Account 类型上的多态。我们正在使用 Scala 的类型系统编码领域逻辑，同时在函数定义中编码了只有某种账户类型才能计算利息的逻辑。接下来看一下它们是怎么工作的，以及从长期来看这个策略给我们带来了什么：

- 领域逻辑现在更加明确。一个编码了账户类型的单独数据类型使领域模型有了更好的表现力。
- 函数 calculateInterest 将合法的账户类型作为签名的一个部分——限定泛型类型 A 作为 InterestBearingAccount 的子类型。该限定将成为程序文档的组成部分。因此 API 的用户不用看具体的实现就知道函数允许的合法账户类型。
- 再也不用写测试用例来检验传递给函数的账户类型是否正确。编译器替我们把这件事做了。除了 InterestBearingAccount 和它的子类型，永远不可能用其他类型的账户来调用这个函数。
- 函数唯一能接受的就是账户信息，而它们是 InterestBearingAccount 的子类型，通过输入这些信息，可以确保编译器在处理时拥有更多信息。于是它可以基于此类信息进行优化。

这个例子向我们展示了如何通过类型的力量使领域模型更加丰富多彩并且更加简洁明了。我们获得了更好的表达性，通过将大量测试委托给编译器来进行，甩掉了多余的“脂肪”，同时，还解决掉了领域逻辑校验中的样板文件。在本书中，将越来越多地使用类型领域模型，探索这种方式给实现带来的所有好处。

关于类型系统的说明

2.2 节刚刚向我们揭开了类型系统的面纱，让我们看到了一个强有力的类型系统如何在领域模型中实现不变性和约束，从而在编译器中获取更多的帮助。在清单 2.1 的 calculateInterest 例子中，我们传入了一个具体数据类型 Account，并在同一个函数中为所有不同类型的账户处理了所有的逻辑。如果在清单 2.2 中运用类型系统的力量，就限制了 Account 的数据类型，这样函数体中只包含针对 InterestBearingAccount 计算利息的逻辑。编译器现在拥有额外的信息，这样函数只能被产生利息的账户所调用；它将丢弃其他类型账户的调用请求，这也意味着编译器的搜索空间能够更好地结构化。

2.3 领域行为的纯函数

在第 1 章中，已经看到了用纯函数对领域行为建模的优点。通过函数组合，可以用小函数构建更大范围的抽象。借助小型的可重用组件，可以有机地演进领域模型。在 Scala 中，可以编写纯函数，就像在第 1 章中的例子那样。

我们用 `calculateInterest` 作为首发案例继续探索。先准备一些基础的抽象，在领域模型中将会用到（参见清单 2.3），包括为 `calculateInterest` 准备的原型。要留意的是函数 `calculateInterest` 返回 `Try[BigDecimal]` 来处理利息计算失败的场景。如果需要复习 Scala 中 `Try` 是做什么的，可以参见第 1 章 1.3.2 节前的内容。

清单 2.3 基础抽象和 `calculateInterest` 函数的例子

```
trait Account {                                    ← 领域实体 Account 的基础抽象
  def number: String
  def name: String
                                                   ← 可能包含其他属性
}
case class CheckingAccount(..) extends Account     ← 用代数数据类型将支票与储蓄建模为不同的类型

trait InterestBearingAccount extends Account {
  def rateOfInterest: BigDecimal
}
case class SavingsAccount(..) extends InterestBearingAccount
case class MoneyMarketAccount(..) extends InterestBearingAccount
trait AccountService {
  def calculateInterest[A <: InterestBearingAccount](account: A,
    period: DateRange): Try[BigDecimal] = {
                                                   ← 略去实际的逻辑
  }
}
```

接下来，将看到如何对 `calculateInterest` 运用函数组合在系统中建立更强大的功能（参见清单 2.4）。代码片段基本上是独立的——主要目的还是展示函数组合的不同特点，使我们可以在完全纯粹和引用透明的情况下构建领域行为。

清单 2.4 组合导致的抽象进化

```
val s1 = SavingsAccount("dg", "sb001", 0.5)
val s2 = SavingsAccount("sr", "sb002", 0.75)
val s3 = SavingsAccount("ty", "sb003", 0.27)       ← ❶ 储蓄账户例子清单

val dateRange = ..

List(s1, s2, s3).map(calculateInterest(_, dateRange))   ← ❷ 对储蓄账户 list 映射并计算每个余额
```

```
List(s1, s2, s3).map(calculateInterest(_, dateRange))
  .foldLeft(BigDecimal(0))((a, e) => e.map(_ + a).getOrElse(a))

List(s1, s2, s3).map(calculateInterest(_, dateRange))
  .filter(_.isSuccess)

def getCurrencyBalance(a: Account): Try[Amount] = ..

def getAccountFrom(no: String): Try[Account] = ..

def calculateNetAssetValue(a: Account, balance: Amount): Try[Amount] = …

val result: Try[(Account, Amount)] = for {
  s <- getAccountFrom("a1")
  b <- getCurrencyBalance(s)
  v <- calculateNetAssetValue(s, b)
  if (v > 100000)
} yield (s, v)
```

❸ 在储蓄账户list中找到累计总利息

使用过滤器获取余额计算的列表 ❹

❺ 如果净资产值大于100,000，就记录账户号与净资产值

再说一遍，与实际领域模型相比，例子都是非常简单的。但简化有利于在案例中更加清晰地进行阐述。让我们来看看这些案例所演示出来的函数组合的优越性。理论上，可以在 PERL 中尝试这些案例，看看这些组合器是如何工作的。每个返回的那些特定值都可以再次组合，以建立更高程度的抽象。

在清单 2.4 中的第一个例子 ❷ 中，我们用 map 组合器操作在 ❶ 处定义的账户清单。通过使用 map，只需要指明对 list 中的元素做什么，而不需要考虑如何在 list 中迭代。能看出这和命令式编程中的 for 循环有什么区别吗？在 for 循环中，必须明确定义如何在 list 上进行迭代以及每个迭代的元素应该做什么。但函数式组合器如 map，仅仅需要定义“做什么”部分就可以了。而“如何迭代”已经在组合器中抽象了。这降低了用户的学习成本，同时也使 API 更容易理解。

第二个例子 ❸ 展示了 fold 组合器，这里用的是 foldLeft。它帮助我们在账户的 list 中进行迭代并计算利息累计的和。它同样是一个纯表达式，并根据每个迭代步骤的累计值做赋值。

下一个例子 ❹ 演示了另一个组合器：filter。这个表达式获取计算利息的列表，不包括任何计算失败的利息。

如果是函数式编程的初学者，我建议在这里停一下，然后试着用命令式语言比如 Java 或者 C# 实现同样的功能。之后就会发现，在这些语言里，会把计算所有利息和的逻辑实现得像一个语句的序列，一个接着一个执行。可能会用 for 循环来遍历所有账户并依次对它们调用函数。在这个处理过程中，我们是在一个可变状态中进行累加。看到这里就会发现，第一个不同点就是使用纯函数时，逻辑看上去是一

个单一的表达式（而不是一个语句的序列）。因此，可以将值从一个函数传递给另一个函数，而不需要任何的中间状态。这也是编译器能够应用某些优化技术（如融合）的原因，这将在下一节中讨论。一个表达式产生一个值，这个值直接提供给另一个表达式。清单 2.4 中，map 组合器从 List(s1,s2,s3) 直接获取输入，不需要任何中间计算，而如果用命令式语言的 for 循环就会需要中间过程。

在清单 2.4中最后的例子 ❺ 是一个更强有力的面向表达式编程的演示。通过碎片化的方式建立表达式，形成最终解决方案，不需要请求任何中间状态。如果依然不确信，那就看图 2.2，它显示了两者之间的转换。另一方面，在命令式语言里，语句伴有大量的副作用，为达成同样的结果，我们所做的将在序列中产生副作用。这个处理很有可能是错误的，因为牵扯到可变状态——因此，函数式编程也通常被称为*面向表达式编程*。图 2.2 列出了语句与表达式之间的区别。

语句	表达式

`if <condition> {`
`  <statement1>`
`} else {`
`  <statement2>`
`}`
	`val result = if <condition> {`
	`  <expression1>`
	`} else {`
	`  <expression2>`
	`}`

• 副作用——每个语句都是一个任务指派 • 不返回任何值——只有副作用 • 难以组合	• 如果每个表达式都引用透明那么整体就引用透明 • 如果表达式返回值的话，返回值可以传递给另一个函数 • 因为面向表达式，所以能够很容易地被组合 • 更容易被推导

图 2.2 语句与表达式的区别。这个汇总清楚地解释了为什么在考虑函数组合时，应该选择表达式而不是语句。

清单 2.4 中最后的例子 ❺ 也描述了 Scala 中的 for 表达式。一个 for 表达式使用了 flatMap 和 map 组合器将操作序列化。序列化被认为是命令式语言的风格，就像刚才所见。事实上，在这个案例中的操作序列化只是一个表达式语法的小糖块。序列操作被翻译成一个等价的单一表达式，就如同用 flatMap 和 map 所做的事。这对我们来说是一箭双雕的好事：对用户来说序列化操作更符合他们的直觉，而面向表达式的执行赋予我们所有函数式编程的好处。图 2.3 将 for 表达式“脱糖”转化成一个 map 和 flatMap 的等价序列。

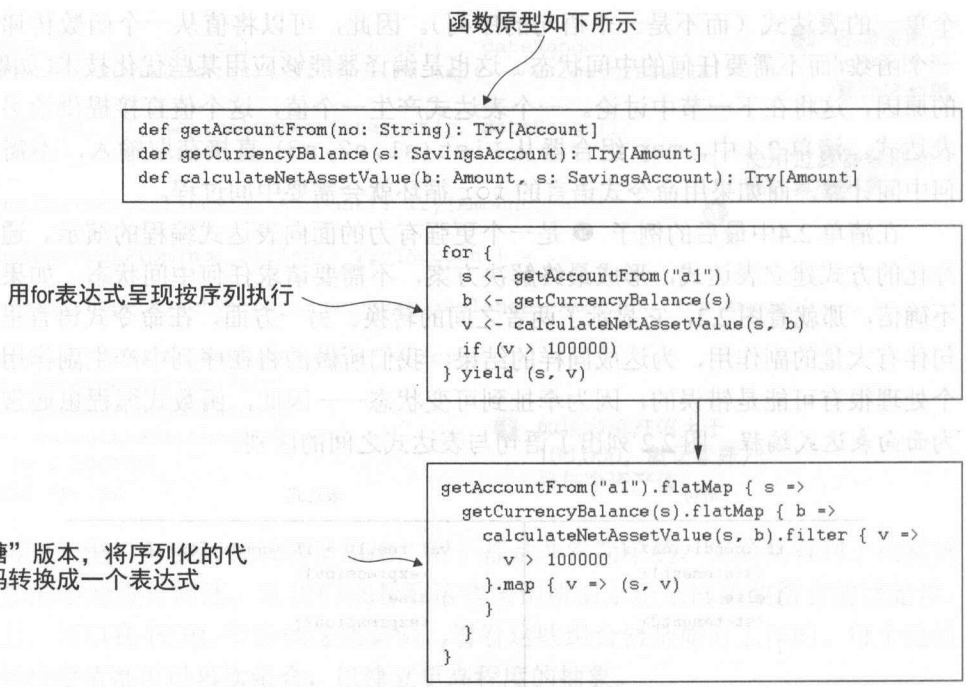


图 2.3 使用 for 表达式的操作序列只是语法上的小糖块。for 表达式可以用 map、flatMap 和 filter 来转换。操作序列被转换成一个能组合的表达式。

事实上，for 表达式是 flatMap 和 map 链条顶端的语法技巧。大家可以在 Martin Odersky 等人编著的 *Programming in Scala*（第 3 版，Artima Press, 2016 年）中学到更多关于 Scala 的 for 表达式的细节内容。但我们的兴趣点是通过使用 Scala 中这种函数式编程的可组合结构，开发对用户来说更具有表现力的模型行为，同时它们也是更加健壮的抽象。

我们可能会担心，如果 for 表达式中的任意步骤发生了失败或异常，这时会怎么样。不用担心，这些都会在 map 和 flatMap 组合器的实现中处理掉。如果有步骤失败了，整个序列会自动中断。所以，异常处理也是被这些抽象组合照顾到的另一个方面。

2.3.1 回顾抽象的纯粹性

大家可能注意到了本书在对领域行为建模时使用了术语*纯函数*。[1] 在第 1 章也解释了纯粹性如何帮助我们推导函数。再强调一遍，一个函数如果没有副作用，那么它就是纯函数。那么什么是副作用呢？副作用就是不在实现的函数控制范围之内的

1 我们在第1章的1.3节已经学过了纯抽象的优点。

事物。如果在函数中操作了文件系统，或者用到了数据库或其他任意的外部资源，那么就是在制造副作用。对纯函数的定义是，每次用相同的输入调用它，都期望得到相同的输出。事实上，纯粹的代码还有一个正式的名字：*引用透明*。它也是在说同一件事：给一个表达式传入相同的输入将得到相同的输出。

本节会介绍结合纯函数和抽象，如何对模型进行优化。让我们从个人银行业务模型的一个例子开始。考虑两个功能：根据一个账户中的余额计算利息，以及基于某种逻辑从计算出来的利息中扣除税费。同样，特定的逻辑在这里并不是很重要，可以考虑一些虚构的最简化的假设。清单 2.5 将用基本抽象实现这两个功能。

清单 2.5 纯粹的力量

```
def calculateInterest: SavingsAccount => BigDecimal = { a =>
  a.balance.amount * a.rateOfInterest
}

def deductTax: BigDecimal => BigDecimal = { interest =>
  if (interest < 1000) interest else (interest - 0.1 * interest)
}

trait Account {
  def no: String
  def name: String
  def balance: Balance
}
case class SavingsAccount(no: String, name: String,
  balance: Balance, rate: BigDecimal) extends Account

case class Balance(amount: BigDecimal)

val a1 = SavingsAccount("a-0001", "ibm", Balance(100000), 0.12)
val a2 = SavingsAccount("a-0002", "google", Balance(2000000), 0.13)
val a3 = SavingsAccount("a-0003", "chase", Balance(125000), 0.15)

val accounts = List(a1, a2, a3)

accounts.map(calculateInterest).map(deductTax)

accounts.map(calculateInterest andThen deductTax)
```

❶ 集合上的两次 map

通过组合函数融合 map，在这里用了“andThen”，f andThen g 等同于 g(f(x))，还有另外一种组合器，“compose”——f compose g 等同于 f(g(x))

❷

现在已经熟悉了清单 2.5 中代码所做的事。在第二个 map 表达式中所用的技术被称为融合（*fusion*）：通过使用函数组合，“融合”了两个 map 组合器。在计算扣税后的净利息上，表达式 ❶ 和 ❷ 之间是否有什么区别？

让我们聚焦在函数 calculateInterest 和 deductTax 的定义上。对于变体 ❶，它用 calculateInterest 和 deductTax 做了两次 map，这种行为如图 2.4

所示。第一个 map 生成一个余额的 list 并作为第二个 map 的输入，第二个 map 生成扣税后的净利息列表。因此，在输入的集合与最终输出之间，有一个中间状态的集合，也就是利息的列表。

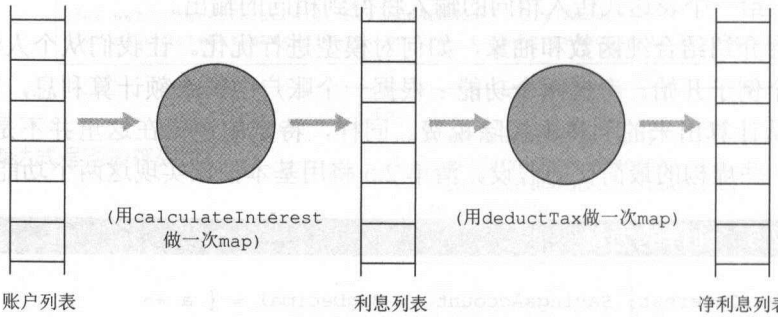


图 2.4 在一个集合上 map 两次，生成最终的净利息列表。这里产生了一个中间状态集合，如果这个集合的空间很大就有可能产生问题。

在第二个变体 ❷ 中，做了优化。注意这两个函数的类型：SavingsAccount => BigDecimal 和 BigDecimal=>BigDecimal。它们排好队以便于组合。既然不再是在列表上独立地应用两个函数，为什么不直接应用组合本身呢？毕竟那是我们正在第一个变体上所做的事：将第一个应用的结果传递给第二个应用，在这个过程中生成一个中间状态集合。图 2.5 展示了组合的方式。

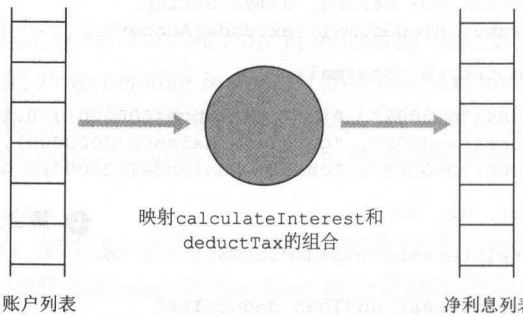


图 2.5 通过 map 组合函数干掉了图 2.4 中的中间状态集合。

组合带来了性能上的优化。在代码中会看到 list.map(f).map(g) 形式的模式，可以用 list.map(g compose f) 替换它们以达到最优化。这个语句假设可以对 f 和 g 使用这种优化。但对于 Scala，这是一个合理的建议吗？

想象一下如果选择一个将产生副作用（比如，它将创建一个文件或更新数据库）的 f，以及将使用这些副作用结果的 g 会发生什么？这个优化技巧会挂掉。函数 f 和 g 不再仅仅依赖于它们所取得的输入（就像之前的例子 calculateInterest 和 deductTax），同时还依赖于外部资源。它们不再是引用透明，也失去了纯洁性

和组合能力。因此，从这个讨论得出的结论是要尽可能地使函数不受副作用的影响。不要将纯逻辑和副作用捆绑在一个函数里，解耦它们，这样至少可以在其纯逻辑部分获取组合性的红利。

2.3.2 引用透明的其他好处

在前面的章节中，已经了解了如何利用融合技术来组合纯函数并且执行优化。使函数引用透明还有另外两个方面的好处，这是函数式编程的普适优点，不是仅仅针对 Scala。

易测性

纯函数是很容易测试的，因为不依赖于任何副作用或外部状态。如果函数只依赖于指定的输入，那该规范也能被描述为属性。可以把这些属性作为公式提供给基于属性的测试库，诸如 ScalaCheck（http://scalacheck.org），它们会生产随机数据来替我们测试。相对于传统的基于 xUnit 测试来说，这是个巨大的优势，而且更有效率。在本书中还将学到更多基于属性的测试。

并行执行

如果代码不受副作用的影响，可以更有效率地使用并行数据结构，而不用担心外部状态会不会下绊子。在 Scala 中，如果代码片段中对一个集合进行了映射，如 `collection.map(//..`，只须将其改为 `collection.par.map` 就能转化成并行执行。这将给我们带来希望的效果，同时还没有任何不良反应，只要传给 `map` 的是一个纯函数。

2.4 代数数据类型与不变性

在书中讨论领域模型时，是在说关于实体、值对象以及其他类型抽象的建模，而代数数据类型（ADT）也是将要遇到的一个概念。需要对在 Scala 中能够建模的各种类型的 ADT 有一个清晰的认知。我们都知道数据类型是什么，但在什么场景下是代数数据类型？

在本节中，不会先开始讨论理论。相反，会从一个例子开始，展示 ADT，以及它们如何帮助我们对领域进行建模。

2.4.1 基础：和类型与乘积类型

在个人银行业务领域中，需要处理不同类型的货币——USD（美元）、AUD（澳元）、EUR（欧元）和 INR（印度卢比）。虽然有不同的分类，但它们的基本类型依

然是货币。在清单 2.6 中展示了如何在 Scala 中对它们进行建模。

清单 2.6 货币建模（和类型）

```
sealed trait Currency
case object USD extends Currency
case object AUD extends Currency
case object EUR extends Currency
case object INR extends Currency
```

在这里使用了一个基础抽象来生成货币模型。还定义了子类型用来识别系统中不同类型的货币。在模型中，货币的实例可以是以下的任意一种：USD、AUD、EUR 或 INR。它只能是其中的一个值，不能有一个货币既是 USD 又是 INR。所以这里是“或”的关系，然后通过加号提供一个“或”逻辑：type Currency = USD + AUD + EUR + INR。

于是有了一个新的数据类型：Currency。能找出类型 Currency 有几种独立的值吗？用类型理论的术语来说，我们将其称为数据类型 Currency 的“居民”（inhabitant）数。答案是 4，同时也可以发现，它就是 Currency 数据类型能拥有的独立值数量的和。所以，Currency 就是一个和类型（sum type）。

让我们看另外一个例子，这次它是来自 Scala 标准库的例子。清单 2.7 展示了 Scala 如何建立“二选一”（either）数据类型。

清单 2.7 Scala 中的 Either 数据类型

```
sealed abstract class Either[+A, +B] { //..
}
final case class Left[+A, +B](a: A) extends Either[A, B] { //..
}
final case class Right[+A, +B](b: B) extends Either[A, B] { //..
}
```

清单 2.7 展示了 Either 数据类型的基础模型，它获取两种类型的参数，同时还有 Left 和 Right 两个特化（specialization）。当构建一个 Either 的实例时，可以用 Left 的构造器注入一个类型 A 的值，或者用 Right 构造器注入一个类型 B 的值。因此当定义一个 Either 的实例时，它要么是 Left，要么是 Right，不可以两者都是。这是和类型的另外一个例子。

测验时间 2.1 对于清单 2.7 中 Either[A, B] 数据类型，它的居民数是多少？提示：不是 2。

回顾一下我们早些时候讨论的 Account 抽象。清单 2.8 中展示了 Scala 中 Account 类一个细微改动的版本。

清单 2.8 Account 抽象（乘积类型）

```
sealed trait Account {
  def number: String
  def name: String
}

case class CheckingAccount(number: String, name: String,
  dateOfOpening: Date) extends Account

case class SavingsAccount(number: String, name: String,
  dateOfOpening: Date, rateOfInterest: BigDecimal) extends Account
```

一个 Account 可以是 CheckingAccount，也可以是 SavingsAccount。这是另一个和类型的例子。但现在让我们关注一下在一个 Account 特定实例的内部会有什么。一个 CheckingAccount 有一个 number、一个 name 和一个 dateOfOpening。将这些属性捆绑在一起创建了一个新的数据类型，这样就可以给这个数据类型的集合赋予新的语义。在类型语言中，将这个表达成 (String, String, Date) => CheckingAccount，或者更通用一点，表达为 type CheckingAccount = String x String x Date。简单地说，CheckingAccount 数据类型是所有合法数组 (String, String, Date) 组合的集合，它就是这 3 种数据类型的笛卡儿积。因此也被称为乘积类型。因此在这个例子中，Account 是和类型，而每个类型的 Account 都是一个乘积类型。图 2.6 描述了和类型、乘积类型以及它们的居民数。

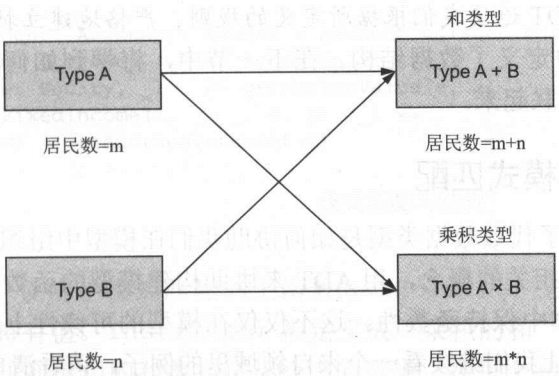


图 2.6 和类型与乘积类型的概要图。每个类型的居民数决定了和类型与乘积类型中的居民数。

测验答案 2.1 你可以通过将每个独立类型的居民数相加得到最终的居民数。在 Currency 的例子中，所有独立类型都只有一个居民，因为在 Scala 中它们每个都是一个 case 对象，用单例的方式建模。在 Either[A, B] 的例子中，它的居民数

取决于 A 和 B 中的居民数。比如，如果有一个 Either[Boolean, Unit]，那么居民数应该是 2（Boolean 有 2 个值）+1（Unit 有 1 个值），所以是 3。这个问题可能会有一点难度。

2.4.2 模型中的 ADT 结构数据

和类型与乘积类型为领域模型中不同数据的结构化提供了必要的抽象。和类型使我们可以将变化建模在一个特定数据类型内部，乘积类型帮助群组相关的数据成为一个更大的抽象。回想一下我们讨论的函数式编程是如何通过小型抽象创建更大的抽象的。这正是一个适当地应用 ADT 使得数据类型成为可组合的例子。

case class CheckingAccount(number: String, name: String, dateOfOpening: Date) 就是组合了一个 (String, String, Date) 并给它打上了一个新的数据类型的标签，CheckingAccount。闭上眼睛想一下：我们可以避开额外的标签跟三个元素的数组一起工作，但有了标签的数据类型，就可以组合一个被命名过的特性，它也会映射到我们正在建模的领域。

使用 ADT 的另一个好处是编译器会自动校验包含合法数据类型的各种标签。我们试着用一个额外的项目 rateOfInterest: BigDecimal 来实例化 CheckingAccount，编译器会立刻捕获到这种行为。生产的每个实例必须是根据数据类型所拥有的列表进行合法地编码——而不仅仅是标签合法（在我们的例子中就是 CheckingAccount 和 SavingsAccount），不过组件数据类型还必须满足一对一。本节的结论就是 ADT 迫使我们根据所定义的规则，严格地建立抽象。

ADT 在模型中定义了数据结构。在下一节中，将学到如何将领域特定行为和 ADT 的指定标签关联起来。

2.4.3 ADT 与模式匹配

现在已经知道了代数数据类型是如何协助我们在模型中组织数据结构的，接下来看一下另外一个相关的概念，用 ADT 来协助构建模型的函数性。模式匹配有助于在各个 ADT 变体中保持函数性。这不仅仅在模型的可读性上进行了提升，还能使代码更加健壮。让我们继续看一个来自领域里的例子，请看清单 2.9。

清单 2.9 中定义了一对和类型与乘积类型来对领域对象进行建模。我们将用这些 ADT 解释如何将领域行为和每个 ADT 结合起来。

清单 2.9 ADT 中的模式匹配

Account 的抽象与前面所实现的类似

```
case class Account(no: String, name: String, dateOfOpening: Date,
  balance: Balance)

sealed trait Instrument

case class Equity(isin: String, name: String, dateOfIssue: Date)
  extends Instrument

case class FixedIncome(isin: String, name: String, dateOfIssue: Date,
 issueCurrency: Currency, nominal: BigDecimal) extends Instrument

sealed trait Currency extends Instrument
case object USD extends Currency
case object JPY extends Currency

case class Amount(a: BigDecimal, c: Currency) {
  def +(that: Amount) = {
    require(that.c == c)
    Amount(a + that.a, c)
  }
}

case class Balance(amount: BigDecimal, ins: Instrument, asOf: Date)

def getMarketValue(e: Equity, a: BigDecimal): Amount = //..
def getAccruedInterest(i: String): Amount = //..

def getHolding(account: Account): Amount = account.balance match {
  case Balance(a, c: Currency, _) => Amount(a, c)
  case Balance(a, e: Equity, _)   => getMarketValue(e, a)
  case Balance(a, FixedIncome(i, _, _, c, n), _) =>
    Amount(n * a, c) + getAccruedInterest(i)
}
```

Instrument 的抽象表示成和类型与乘积类型的组合

辅助类

Balance 的抽象存储了用户不同类型 Instrument 的余额

针对一个特定的 Instrument 类型（Currency 与 Equity）的模式匹配

模式匹配与结构

在清单 2.9 中，使用了模式匹配的主函数 `getHolding`，它根据 `Balance` 的类型计算账户的净持有值。`Instrument` 被定义成“乘积的和”类型，同时模式匹配使我们可以通过正在使用的和类型执行恰当的逻辑。在每个和类型内部还有乘积类型，这也能通过模式匹配来解构。这将确保我们可以取到必要的属性值用于计算。如果使用 OO 或子类型来编码这类实现，就会将我们引向观察者模式[1]，我们也都知

1 如须了解观察者以及其他设计模式的细节，请参考Erich Gamma等人所著的*Design Patterns: Elements of Reusable Object- Oriented Software*（Addison-Wesley Professional, 1994年）。

道，这会面临很大的风险。[1]

Scala 模式匹配的另一个巨大的好处是编译器会检查模式匹配的穷尽性。如果忘记了在模式匹配条款中插入任何 Balance 的计数，编译器会丢个警告出来。如果在和类型中添加了一个额外的计数，编译器会指出所有需要用新增的 ADT 变体更新模式匹配选项的地方。

2.4.4 ADT 鼓励不变性

Scala 代数数据类型的一个基本特质就是鼓励不变性。在编写 `case class Person (name: String, address: String, dateOfBirth: Date)` 时，Scala 默认创建了一个不可变的抽象。如果需要使属性具有可变性，必须对该属性用 var 做明确的定义来声明需要可变性。不过我们不会这样做，因为函数式编程使抽象应用透明，鼓励不变性，避免原地突变。

那么问题是，如果在 Scala 中定义了一个 ADT，如何改变它的属性值呢？答案是不能改变。实际上，我们将根据原始的抽象创建额外具有修改后的值的一个抽象（参见清单 2.10）。

清单 2.10 Scala 的 case class，默认是不变的

```
case class SavingsAccount(number: String, name: String,
  dateOfOpening: Date, rateOfInterest: BigDecimal) extends Account

val today = Calendar.getInstance.getTime

val a1 = SavingsAccount("a-123", "google", today, 0.2)
val a2 = a1.copy(rateOfInterest = 0.15)
```

原始的账户

用变更值替换

在清单 2.10 中，创建一个 SavingsAccount 的实例 a1，作为今天开户的账户。然后，将利率从 0.2 改到 0.15。用 Scala 中的方式就是利用 copy 方法，用新的 rateOfInterest 值创建另一个 SavingsAccount 实例，这样也就不会违背不变性。

---

1 Java中的观察者模式会导致代码结构变得很复杂，非常难以扩展。这也可能是为什么这个模式会有非常多的变种。如须了解细节，可参考Martin E. Nordberg III所著的*Variations on the Visitor Pattern*（http://c2.com/cgi/wiki?VariationsOnTheVisitorPattern）。相比较而言，代数数据类型解决该问题的方式更加优雅。

测验时间 2.2 假设有如下 ADT 定义：

```
case class  Address(no: Int, street: String, city: String, zip: String)
case class Identity(name: String, address: Address)
```

同时还用以下方式创建了一个 Identity 实例：

```
val i = Identity("john", Address(12, "Monroe street", "Denver", "80231"))
```

如何在不直接修改的前提下，将邮政编码（ZIP code）改为 80234？提示：使用嵌套副本。

不变性拥有非常多的优点，其中很多已经在第 1 章中讨论过了。它使并行以及并发设置变得十分简单，可以随意地共享值而无须承受任何来自可变状态的担忧。[1] 不可变数据是函数式编程中实践的最重要的原则之一，与此同时，它与函数纯洁性的思想也可以良好地结合。在领域模型中，要尽可能地努力实现这种思想，保持领域行为的纯洁性，尽量运用不可变数据。在关于实现函数式领域模型的讨论中，会接触到建立在 ADT 之上更高阶的抽象，以及用来操作它们的简单好使的组合器。

测验答案 2.2 需要在两个水平上使用 copy 时，这里有一点点小技巧。下面是解决方案：

```
i.copy(address = i.address.copy(zip = "80234"))
```

我们已经看到 Scala 是如何支持函数式编程的基本思想的——函数作为头等值、高阶函数、函数的纯洁性，以及数据的不变性，这些都使 Scala 成为适合实现函数式领域模型的语言。在最后一节中，我们将了解模块化的内容，这将有助于通过碎片化的方式管理整个架构，使模型可以增量式地进化。

2.5 局部用函数，全局用OO

一个非传统的领域模型天生会拥有很多抽象，包括实体、值对象、服务以及相关的行为。如果所有这些产物都归拢在同一个命名空间里，而且需要处理这样一整块巨大的应用时，你会有什么样的感受？把一个模型实现成一个巨无霸的结构极有

1 如须进一步了解更多细节，参见Brian Goetz的著作*Java Concurrency in Practice*（Addison-Wesley Professional, 2006年）。

机会被视为添乱。没有任何清晰的职责划分，抽象也不会被拆分到合适的命名空间，同时也不会有任何基于边界上下文的隔离。毫无疑问，这根本就是一团糟。

模块是所有这些问题的答案。不同的功能需要划分到不同的模块。在个人银行业务系统的上下文环境中，用户的投资组合报告可以成为一个模块，税费计算可以成为一个模块，同时，审核也可以成为一个模块。显而易见，这里必须问的问题是，这些互相交互的模块应该长成什么样。不管如何，整个模型必须作为一个整体来工作，甚至可能需要一些跨越模块的交互。当一次交易发生在在线用户交互模块时，该信息必须流入审核模块。所以，就一定要想清楚，如何通过清晰的职责划分，将模型模块化。

模块必须松耦合高内聚。什么意思呢？一个模块执行一个明确的任务，因此在其内部必须是紧密聚合的[1]。抽象需要做到小而紧凑，它们中的任何一个都只专注在完成某一个特定的事情上。而在另外一方面，当我们讨论两个不同的模块时，两者之间的耦合度应该尽可能小。如果两个模块之间存在强依赖，毫无疑问不是健康的状态，因为一个发生改变就会影响另外一个，这也是违背模块化设计原则的。

2.5.1 Scala 中的模块

Scala 提供特征（trait）和对象作为模块化设计的实现技术。通过 trait，可以进行基于 mixin 的组合。[2] 我们可以用 trait 将几个小型的抽象组合成一个大型的抽象。这些不是函数，这里也不是在谈论函数组合。通常来说，一个 trait 是一个小型的功能单元，它可能包含一个或几个方法，而方法也只用于实现相应的功能。这里有一个领域中的例子：

```
trait BalanceComputation
trait InterestCalculation
trait TaxCalculation
trait Logging

trait PortfolioGeneration extends BalanceComputation
  with InterestCalculation with TaxCalculation with Logging {
  //.. implementations
}
```

前面提过，客户的投资组合是一个独立模块，但它由一些截然不同的功能组成，而且它们也都是完全独立的。所以我们将它们分成独立的 trait（这样它们可以被独立地重用），然后将它们混合在一起组成更大的投资组合功能。需要十分注意的一点是，所有混合的 trait 是彼此正交的——比如说，Logging 可以在很多其他上下文中重用，BalanceComputation 可以被重用为客户账单的组成部分。现在已经为

1 也就是说不应该有部分功能由外部模块来执行。——译者注

2 更多细节参见*Programming in Scala*。

投资组合组合了必要的功能，接下来可以将模块具体化成一个对象：

```
object PortfolioGeneration extends PortfolioGeneration
```

在这里大家可能会问，我们需要特征 PortfolioGeneration 吗？我们不能直接从组合的 mixin 直接实例化对象。在执行具体实现之前，让最终的模块成为 trait 的形式，会是比较常见的优秀实践。明天我们很可能需要定义一个使用 PortfolioGeneration 的更大的模块，那么就可以用中间态的 trait 来混合其他功能。所以，这就是关于模块的模块性与组合性的所有内容。之前看到的只是 Scala 模块化技术中最简单的一个。Scala 的类型系统具有足够的能力，在这里可以使用很高雅的技巧来实现参数化的模块。还能将一些抽象作为 trait 里的参数，并且在创建最终对象时只生产具体实例。这是非常有用的技术，本书将结合领域中的例子来演示它的威力。

在一个领域模型的实现中，一个模块实现一个特定业务功能。比如，客户的税费计算可以成为一个模块，投资组合的生成可以成为另外一个模块。一个模块通常是一个边界上下文的一部分，一个边界上下文可以包含很多模块。

模块需要能够被参数化，以便根据业务规则包含不同的变体。比如想要计算一个账户在一个指定时间段的累计利息，在实现的例子中，就算忽略非常细节的业务复杂逻辑，也有以下部分需要计算：

1. 利息的计算。
2. 需要从利息中扣除的税费。
3. 第二项需要详细的税表，它包含具体交易类型所对应的缴税项目和税率。（举个例子，在这里把利息计算当作交易类型，而税费将作为其他的交易类型来计算，比如外汇交易或奖金计算。）

定义模块时，比较微妙的部分是要考虑依赖于业务规则的多样性部分，这可能会影响到部署和实现。在我们的场景中可以说：一个计算客户净利息的模块需要涉及另一个计算扣减税费的模块。与此同时，计算税费的模块需要根据税表被参数化，其依赖的是执行的交易类型（在这个例子中是利息计算）。清单 2.11 中展示了这三种模块的定义以及它们的相关性。

清单 2.11 利息计算模块以及它们的依赖

```
sealed trait TaxType
case object Tax extends TaxType
case object Fee extends TaxType
case object Commission extends TaxType

sealed trait TransactionType
case object InterestComputation extends TransactionType
case object Dividend extends TransactionType
```

```
type Amount = BigDecimal
case class Balance(amount: Amount = 0)

trait TaxCalculationTable {                          <-- 基于交易类型参数化的税费计算表
  type T <: TransactionType
  val transactionType: T

  def getTaxRates: Map[TaxType, Amount] = {
    //..
  }
}

trait TaxCalculation {                               <-- 基于税费计算表参数化的税费计算逻辑
  type S <: TaxCalculationTable
  val table: S

  def calculate(taxOn: Amount): Amount =
    table.getTaxRates.map { case (t, r) =>
      doCompute(taxOn, r)
    }.sum

  protected def doCompute(taxOn: Amount, rate: Amount): Amount = {
    taxOn * rate
  }
}

trait SingaporeTaxCalculation extends TaxCalculation {   <-- 针对一个特定地区的税费计算特例，它有额外的税费需要被征收
  def calculateGST(tax: Amount, gstRate: Amount) =
    tax * gstRate
}

trait InterestCalculation {                          <-- 基于税费计算逻辑参数化的利息计算逻辑
  type C <: TaxCalculation
  val taxCalculation: C

  def interest(b: Balance): Option[Amount] = Some(b.amount * 0.05)

  def calculate(balance: Balance): Option[Amount] =
    interest(balance).map { i =>
      i - taxCalculation.calculate(i)
    }
}
```

这时还不会有任何模块的具体实例——现在只有模块的定义以及它们的依赖，这些体现了领域模型中多样性的部分。在这里，可以用 Scala 提供的抽象类型和 val 实现多样性，这使得依赖路径图清晰且容易理解。

模块组合的最后一个步骤是创建模块的实例，这个步骤将实现某个具体场景的功能。在清单 2.12 中，构建了一个具体模块，`InterestCalculation`，它用 `InterestComputation` 作为交易类型为非新加坡用户计算利息。在这里，需要做的就是为所有抽象类型指定值并具体化所有的模块定义。

清单 2.12 一个计算利息的具体模块

```
object InterestTaxCalculationTable extends TaxCalculationTable {
  type T = TransactionType
  val transactionType = InterestComputation
}

object TaxCalculation extends TaxCalculation {
  type S = TaxCalculationTable
  val table = InterestTaxCalculationTable
}

object InterestCalculation extends InterestCalculation {
  type C = TaxCalculation
  val taxCalculation = TaxCalculation
}
```

请注意，在 TaxCalculationTable 和 TaxCalculation 中抽象的类型成员和值成员已经在各个对象声明中具体化了。该策略的核心结论是不需要确定具体的类型或值，直到声明对象。这对模块组合来说真是太棒了。

图 2.7 描述了如何在 Scala 中使用多个参数化模块的组合生成一个模块的具体实例。这里的重点是保持设计的灵活，这样模块间的通信才能最小化、足够灵活并且可以外部注入。

现在对 Scala 如何映射领域建模技术已经有了比较全面的认知。Scala 的 3 个主要特性，静态类型系统、函数式编程以及头等模块支持使得建模过程变得富有成效而且充满效率。如果没有机会体验本章所有例子，可参见图 2.8 的俯瞰图。如果大家非常用功地尝试了所有例子，那这张图也是非常有用的参考。

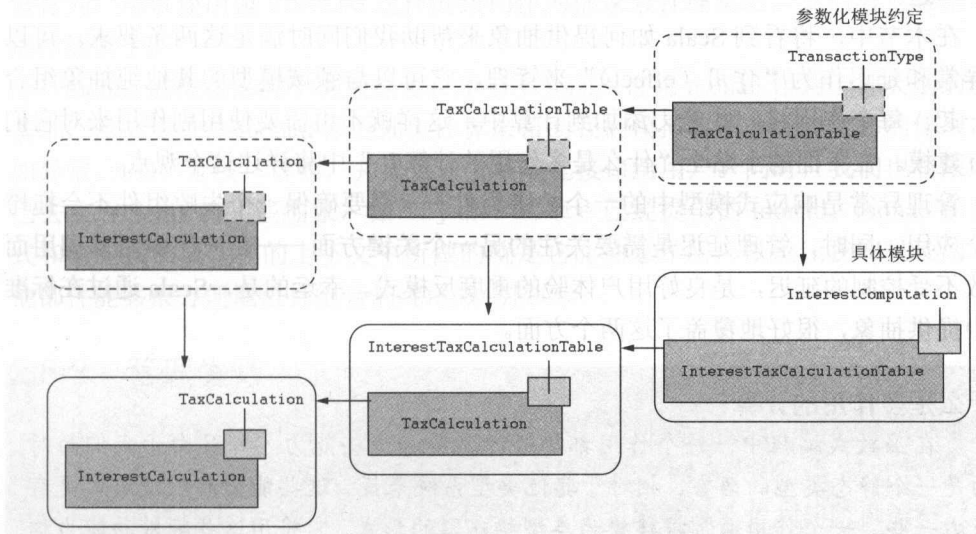


图 2.7 Scala 中的模块演进。使用类型（图中所示浅灰色正方形）参数化模块约定（图中所示深灰色长方形），逐个具体化，并实例化最终的具体模块。

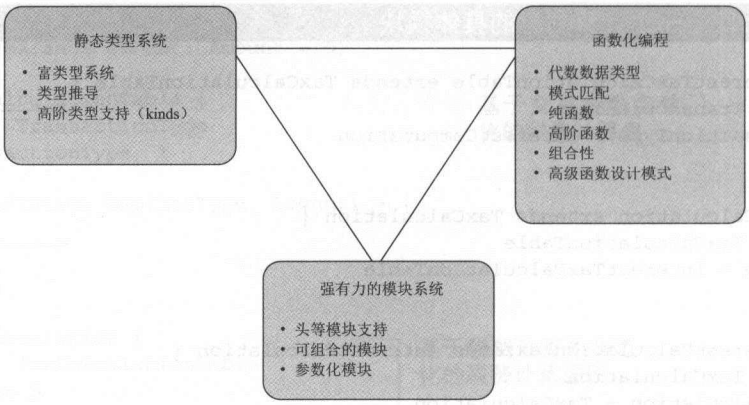


图 2.8 3 个主要特性使 Scala 在领域建模中成为一个成功的平台。

2.6 用Scala使模型具备响应性

使用纯函数的函数化思想与实现是针对领域模型的一个伟大的工程规范。但同样也需要语言的支持来帮助建立一个能够对失败保持足够的灵敏、能随着负载的增长而良好地扩展，同时给用户提供一个友好的体验的模型。第 1 章将此特性称为具备响应性，同时也定义了以下两个需要满足的准则，以便实现模型的响应性：

- 管理失败，也就是针对失败的设计。
- 将长时间运行的处理委托给后台线程，避免阻塞主线程的执行，从而减少延迟。

在本节中，将看到 Scala 如何提供抽象来帮助我们同时满足这两条要求。可以将异常和延迟作为“作用（effect）”来管理，它可以与领域模型的其他纯抽象组合在一起。每个作用将一些能力添加到计算中，这样就不再需要使用副作用来对它们进行建模。在下面的小贴士“什么是多作用的计算？”中将详述这个观点。

管理异常是响应式模型中的一个关键组件——需要确保一个失败组件不会拖垮整个应用。同时，管理延迟是需要关注的另一个关键方面——在应用中阻塞调用而导致不受控制的延迟，是良好用户体验的重度反模式。幸运的是，Scala 通过在标准库中提供抽象，很好地覆盖了这两个方面。

什么是多作用的计算？

在函数式编程中，每个作用都会给计算增加一些能力。并且由于我们面对的是一个静态类型的语言，相对于其他类型系统来说，这些能力的表现形式更有威力一些。一个作用通常被建模为类型构建器的形式，它将用这些额外的能力构

建类型。比如有个类型A，同时想要增加集合的能力，就可以创建一个A的集合作为一个独立的类型。通过构建一个类型List[A]来完成这件事（相应的类型构造器就是List），也就在A上增加了集合能力。类似地，可以用Option[A]来为类型A增加可选性的能力。在下一节中，将学到如何用类型构造器诸如Try和Future分别为异常和延迟的作用建模。在第4章中，将讨论使用applicative和monad的更加高级的作用处理。

2.6.1 管理作用

当讨论异常、延迟以及其他构建模型响应能力的内容时，必须考虑如何将这些概念整合到函数式编程领域中。第1章将这些称为副作用，同时也提醒我们要注意它们给纯领域逻辑所带来的问题。现在将要讨论如何管理这些副作用以便使模型拥有良好的响应性，如何将它们作为模型的一部分来对待，这样它们就能以引用透明的方式和其他领域元素组合到一起。

在Scala中，将这些副作用作为作用来对待，也就是说在容器内抽象它们，然后暴露函数接口给外部。Scala里最典型的将异常当作用来处理的例子就是Try的抽象，就如上文所述。Try提供一个和类型，其中一个变体（Failure）抽象了计算可能抛出的异常。Try在其内部封装了异常的作用，然后提供一个函数化接口给用户。在通常情况下，Try是一个monad。还有一些其他类似作用处理的例子。延迟是另外一个例子，也可以把它当作用对待——不需要将模型暴露于无限延迟的异常行为，可以使用像Future这样的结构作为抽象来管理延迟。我们马上就会看到这样的例子。

monad的概念来源于类别理论。本书不会深入讨论monad作为一种类别的理论基础，而会更关注monad作为抽象计算来帮助我们模拟典型非纯粹行为的作用，比如异常、I/O、延续，等等，并提供一个函数化接口给用户。同时，我们也将限制自己去实现Scala的某些抽象，如Try和Future已经提供的monad。monad在函数式和响应式领域建模的上下文中所做的就是它能抽象作用，以及让我们只使用与其他部件能够很方便地进行组合的纯函数化接口。

2.6.2 管理失败

第1章中介绍了针对失败设计的范式。不管如何设计模型，也不管为寻址异常准备了多少安全措施，失败还是会发生。硬件失败、网络失败、第三方软件失败，甚至自己的软件组件，哪怕已经非常仔细地考虑了所有可能想到的防御策略，但还是发生了失败。因此，作为一个模型设计者，应该采取何种策略来应对这种无孔不

入的失败情况？

在第 1 章中已经说过，将代码和错误检查混杂在一起不是好的解决方案。它很难良好地开展工作，而且从软件工程的观点来看，这是一个很糟糕的实践。核心领域代码会被遮盖在大堆的错误检查代码中。

Scala 提供了两个策略来处理异常：

- 明确一部分代码可以产生异常，用类型系统来协助处理。
- 使用抽象，这样不会将异常管理细节泄露到领域逻辑内部，那么核心逻辑就可以保持函数组合性。

```
def getCurrencyBalance(account: SavingsAccount): BigDecimal
```

这个函数在正常情况下，也称为*快乐路径*，函数将取得余额 BigDecimal。但如果遇到异常呢？在某些情况下，函数将抛出异常，顺便说一下，它们可能在整个函数中根本都没有描述过。在 Scala 中，通过将失败按作用来管理，可以更好地处理这种问题。

让我们看一个领域中具体的例子，以便更好地理解用真实的函数化方式管理异常的实际好处。清单 2.13 展示了 3 个函数，每个函数在特定情况下都会发生失败。我们会让真相大白于天下——这些函数返回 Try[BigDecimal]，而不是 BigDecimal。我们之前已经看过 Try，也知道它在 Scala 代码内部是如何进行抽象的。这里通过返回一个 Try，使这个函数明确地表明它有可能会发生异常。如果一切正常，将从 Try 的 Success 分支获取结果，如果有异常，将从 Failure 变体中取得结果。这里最重要的一点是，异常永远逃不出 Try 的手掌心，不会成为污染代码的有害副作用。这样就达成了 Scala 失败管理两大策略中的第一个承诺：勇敢直面函数会发生失败的事实。

清单 2.13 在 Scala 中管理失败

```
def calculateInterest[A <: SavingsAccount](account: A,
  balance: BigDecimal):Try[BigDecimal] = {
  if (acc.rate == 0) Failure(new Exception("Interest Rate not found"))
  else Success(BigDecimal(10000))
}

def getCurrencyBalance[A <: SavingsAccount](account: A):Try[BigDecimal] = {
  Success(BigDecimal(1000L))
}

def calculateNetAssetValue[A <: SavingsAccount](account: A,
  ccyBalance: BigDecimal, interest: BigDecimal): Try[BigDecimal] = {
  Success(ccyBalance + interest + 200)
}
```

但是组合性的承诺怎么办呢？没事，Try 作为一个 monad 同样可以满足我们，同时还提供了很多高阶函数。比如 Try 的 flatMap 方法使其成为一个 monad 并帮助我们与其他可能产生失败的函数组合起来：

```
def flatMap[U](f: T => Try[U]): Try[U]
```

前面已经看过 flatMap 如何与计算绑定到一起，在帮助我们编写优雅的序列化 for 表达式的同时又不丢弃面向表达式赋值的优点。我们可以通过 Try 组合可能失败的代码来达到同样的效果。

```
for {
  b <- getCurrencyBalance(s1)
  i <- calculateInterest(s1, b)
  v <- calculateNetAssetValue(s1, b, i)
} yield (s1, v)
```

这个代码处理所有异常，并将它们良好地组合在一起，同时用简洁明了的方式描述了代码的目的。这就是在领域模型中应用强有力的语言抽象的好处。Try 是处理异常的抽象，而 flatMap 是程序穿越欢乐路径的秘密工具。所以我们现在拥有了可以抛出异常的领域模型元素——只需要用 Try 抽象来管理作用，就可以使代码对失败保持足够的弹性。

2.6.3 管理延迟

就像 Try 用作用管理异常一样，Scala 库中另外一个抽象 Future 会帮助我们将延迟作为一个作用来管理。这意味着什么？响应式编程建议我们的模型需要对延迟的变化保持弹性，因为系统负载的增长、网络延迟以及其他超出实现所能控制的方面都有可能导致延迟。为了提供一个可被接受并有良好响应时间的用户体验，我们的模型需要在延迟的界限上做一些保证。

方法很简单：将运行时间长的计算封装成一个 Future。这个计算将被分发给一个后台线程，而不用阻塞主线程的执行。结果就是，用户体验没有受到影响，同时只要计算完成后计算结果就对用户可用。要注意的是，在计算失败的情况下，这个结果也就是个失败——因此 Future 同时将延迟和异常作为作用来处理。

Futrue 同样是一个 monad，就像 Try 一样，也有 flapMap 方式来协助我们将领域逻辑绑定到计算的欢乐路径。在第 1 章清单 1.14 中已经提及要将延迟绑定到最慢的计算，而不是每个独立计算的和。就算面对失败，也可以在基于 future 的计算中设定超时，这有助于对每个客户端的 SLA 来约束延迟。

继续我们当前的思路，想象一下，在清单 2.13 中的函数已经包含网络调用，因此对于每次调用都伴有潜在的长时间延迟。就像前面建议的那样，要使用户明确我

们的 API，并使每个函数都返回 Future（参见清单 2.14）。

清单 2.14 在 Scala 中将延迟作为作用管理

```
def calculateInterest[A <: SavingsAccount](account: A,
  balance: BigDecimal):Future[BigDecimal] = Future {
  if (acc.rate == 0) throw new Exception("Interest Rate not found")
  else BigDecimal(10000)
}

def getCurrencyBalance[A <: SavingsAccount](account: A)
➥ :Future[BigDecimal] = Future {
  BigDecimal(1000L)
}

def calculateNetAssetValue[A <: SavingsAccount](account: A,
  ccyBalance: BigDecimal, interest: BigDecimal): Future[BigDecimal] =
  Future {
  ccyBalance + interest + 200
}
```

明确 Future 作为返回类型

通过使用 flatMap，现在可以将功能连续地组合成另一个 Future。最终效应就是整个计算被分发到一个后台线程，主线程的执行保持空闲。用户体验得到了保证，我们也实现了响应式原则的要求——系统对网络延迟的变化保持弹性。清单 2.15 演示了 Scala 中 future 的连续组合。

清单 2.15 future 的连续组合

```
val result = for {
  b <- getCurrencyBalance(s4)
  i <- calculateInterest(s4, b)
  v <- calculateNetAssetValue(s4, b, i)
} yield (v)

result onComplete {
  case Success(v) => //.. success
  case Failure(ex) => //.. failure
}
```

future 完成后的成功路径

future 完成后的失败路径

在这里，result 也是个 Future，我们可以同时为成功或失败路径插入回调。如果 Future 成功结束，就可以取得净资产值传给客户端。如果它失败了，也可以捕获异常并针对异常实现某些处理。而且通过使用 flatMap 的连续组合，Scala 的 Future 提供了许多并发组合器，能够建立多个 future，然后让它们并发地执行，通过异步非阻塞并行代码返回结果。在第 6 章中将讨论如何用 Scala 的 Future 实现并行非阻塞的代码。

2.7 总结

在本章中，我们学到了一些核心特性，正是它们让 Scala 成为一门适合领域建模的函数式编程语言。我们可以用任何想要的语言来做领域建模。但有些语言能够提供更好的语言和库支持，这使我们可以在更高水平的抽象上进行编程。Scala 就是这样一个语言。本章几个结论总结如下。

- *Scala 强有力的类型系统*：Scala 有一个强有力的类型系统，这可以用来高效地对某些领域逻辑进行编码。有效地使用类型系统可以减少样板文件与非必要的测试代码。
- *针对函数式编程的头等语言支持*：Scala 作为一个头等范式支撑函数式编程，它有来自高阶函数和 Scala 标准库中丰富的组合器的支持。使用函数作为头等抽象，能够实现引用透明并且可组合的领域行为。
- *代数数据类型和模式匹配支持*：Scala 支持使用 ADT 实现对不可变数据的建模。这些数据可以通过使用模式匹配与控制逻辑组合在一起。事实上，ADT 和模式匹配的组合提供了一个强有力的手段，用简洁的方式来表达领域逻辑。
- *头等模块*：模块化是软件工程优秀实践的一个关键方面，而 Scala 针对模块提供了强大的支持。Scala 中的 trait 和 object 帮助我们定义了可组合的模块，用小型组件演进出更大型的组件。理想的领域建模实现方式是局部用函数全局用对象，而 Scala 就是这样一种语言。

3 设计函数式领域模型

本章包括

- 设计领域模型——函数式及代数方法
- 将领域的代数从领域的实现中解耦
- 在 API 的设计中强制执行代数法则
- 实现领域对象的生命周期模式

前面的章节谈到了函数式编程和数学，特别是代数。我们已经了解了代数数据类型、和类型以及乘积类型，也知道如何组合它们形成抽象来对领域元素进行建模。本章会在更高的层面继续这个讨论，我们将从模型开始，使用类型的代数组合，为领域模型建立 API。这些 API 将遵守领域的法则，我们将运用类型的代数确保它们被正确构建，还将学到如何按照代数规格开发 API、编写属性，以及检验组成业务规则的领域法则。

我们已经看到 3 种基本的领域对象生命周期模式：聚合（aggregate）、工厂（factory）和仓储（repository）。本章将展示如何用函数式编程的基础风格在领域模型中实现它们。

图 3.1 描述了在不同章节中的学习路径，以及每章能够提升的使用函数方式建立领域模型方面的能力。

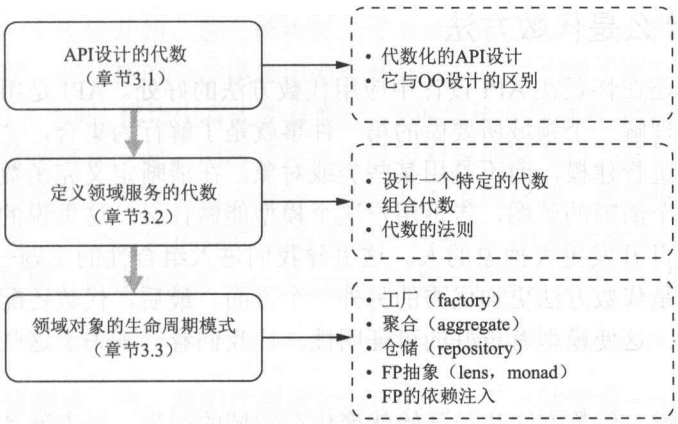


图 3.1 本章学习过程。

3.1 API设计的代数

在使用类似 Java 语言的面向对象（OO）开发中，我们会用*接口*（interface）来做 API 设计，它将最终把模型的规则发布给终端用户。当准备好接口之后，就可以开始类和对象的具体实现。首先定义类，然后加入一些操作作为这个类的方法。在函数式编程中，将这个过程颠倒了一下——首先定义对应到基本领域行为的操作，然后将相关的操作组合成模块，形成更大的用例。每个行为都用基于类型的函数来建模，而类型描述了领域的数据、类或对象。在下一节中，将基于这个函数范式设计一个领域服务。

在函数式领域模型中定义的模块是*函数的集合*，它们操作一系列的*类型*，遵从一系列被称为领域规则的不变式。用数学的术语来说，这就是模块的*代数*。如果对代数这种正式的定义不是很熟悉，本书将在这里重新定义，并且解释每个元素是如何映射到所声明的模块的代数的。

- *一个或多个集合*：在我们的用例中，集合就是数据类型，它们是模型的组成部分。
- *一个或多个函数，操作集合的对象*：在我们的用例中，这些函数就是定义并发布给用户的 API。
- *一些原理或法则被假设为真，且可以被用来得出其他定理*：在 API 中定义了操作时，规则就要详细说明这些操作之间的关系。

要注意到，我们一直都没有讨论实现，因为它不是代数的组成部分，它是行为发布的内容，它组成了 API 的代数。

3.1.1 为什么是代数方法

大家可能还在怀疑在 API 设计中应用代数方法的好处。API 是用户非常关注的内容。用户要理解一个领域所要做的第一件事就是了解行为集合，它们用代数的方式结合纯函数进行建模，而不是用某些类或对象。在清晰定义完函数之后，我们将给用户提供一个清晰的蓝图，告诉用户这个模型能做什么。这里说的用户，是指那些会使用 API 并开发更大抽象的人。这引导我们进入组合性的主题——API 同样需要组合。这也是代数方法更加优秀的另外一个方面。最后，代数还配备了一系列能够核实的规则。这使模型有更好的可证明性。让我们看一下关于这些优点更细节一点的内容。

- 明确清晰：代数方法从一开始就聚焦在领域的行为。行为就是用户所看到的表现，而实现这些行为的函数就是作为建模者用来组合并设计领域模型的函数。对于对象或类不必有认知负担——仅仅是纯函数组合在一起成为实现更大的行为的模块。
- 组合性：一个函数拥有一个代数。当类型匹配时函数就可以组合——如之前所见。将 API 看作函数时，可以通过组合代数本身来建立更大的函数。我们完全不需要知道任意组合函数是如何实现的。对传统 OO 中的类和对象来说，类级别的组合并不是一个定义明确的操作。
- 可证明性：通过定义代数的规则，实现了模型的可证明性。这使模型足够健壮。包含在核心模块实现之内的属性集合能够确保我们不会因为规格上的某些变更就违反它们。

所以这个讨论得出的主要结论就是要重视基于代数的设计理念。一个代数就是一系列类型、一系列函数以及与函数相关的一系列法则的组合。

要注意的是，这里并没有谈论任何关于组成模型 API 的类或函数的具体实现。最初的焦点完全是关于代数以及 API 的组合性方面。同时，这和我们在代数中所做的也很类似——如果有 `y=f(x)` 和 `z=g(y)`，就可以得到一个组合的函数 `g(f(x))`，不用管任何 `f`、`g`、`x` 或 `y` 的具体表现是什么。具体实现会随后以代数解释程序的方式存在于生命周期中。因此我们现在拥有的是一个定义了领域 API 的代数以及多个定义了独立具体实现的潜在解释程序。

3.2 为领域服务定义代数

让我们开始一个案例：为个人银行领域的一个功能子集做基于代数的 API 设计。初始设计应尽可能保持简单。我们将跳过一些可能的难题，晚点再回来解决它们。

让我们从一个模块开始，这个模块定义了 AccountService[1] 的内容以及一些想要支持的操作。在这样的一个模块中会有些什么样的操作例子呢？前文提过，与 OO 基于类的方式不同，模块中首先要有操作，才能使核心领域对象保持瘦小和扁平。行为不再和对象捆绑在一起，同时还可以通过将对象的类型作为它们所代表的代数的内容来独立地演变行为。后面会提到，通过与对象的解耦，这个设计能够使函数的可重用性增强。一个可能的操作就是开设一个账户，开启实体 Account 在领域模型中的生命周期。让我们来定义一个 open 函数，它包含一些开户所需的参数：

```
def open(no: String, name: String, openDate: Option[Date]): ???
```

为了让事情简单一些，我们在函数 open 中包含了一些参数——事实上，真实情况下的参数会比这些多得多。但是，函数的返回类型会是什么呢？它可以是一个 Account 类型，指明这是一个新开设的账户。但随后函数 open 可能因为校验 / 确认的失败而导致函数的失败，作为函数式编程世界中的一个良民，我们不会在抛出异常的同时指望用户来捕获它。在这个例子中，比较明智的做法是返回一个非强制的数据类型，如 Option。如果还想同时提醒用户失败时发生了什么，就需要一个数据类型来保存这个信息。在这里有一些选择的余地：可以使用 Scala 标准库中的 Either[2] 或 Try。但大家能分辨出从 Open 返回一个值，比如 Account，与使用不同类型如 Option、Either 或 Try 做赋值抽象时，有什么区别吗？这听起来有一些复杂，所以让我们深入观察一下函数式编程是如何提供赋值抽象的技巧，并带来更好的组合性的。

3.2.1 赋值抽象

在这里将介绍一个非常重要的概念，书中所有讨论领域行为组合的地方都会用到。假设有一个数据类型 Account 作为函数 open 的返回类型，如果操作成功，将返回一个实例化的 Account 对象。如果失败，既可以返回 null，也可以抛出一个异常。在所有这些场景中，返回的是经过计算的值，同时也是这次赋值的结果。让我们把这种情况和那些返回的数据类型为 Option 或 Try 的情况做个比较。严格意义上来说，Option 和 Try 不是数据类型，它们是类型构造器，而且建立了一个高效的计算。Try 抽象了失败的作用，而 Option 抽象了可选的作用（这也意味着不是必须有一个值）。当返回 Option 或者 Try 时，是在返回一个赋值的抽象。函数的调用者可以决定是从抽象中提取必要的值来进行赋值，还是将其和其他抽象组

1 第1章中定义了该服务为一个领域服务。

2 Either是模型分离的另外一种代数数据类型。Either[A,B]要么包含类型A的值（左边的投影），要么包含类型B的值（右边的投影）。

合到一起。这样的好处是可以获得更多的组合性，在下一章讨论的案例中可以看到这一点。在贴士“Try 作为一个抽象”中有更多细节，Try 不仅仅是一个处理异常的手段，同时还提供了抽象能力，这样我们就可以在赋值之前进行组合，建立更大粒度的抽象来应对失败。

3.2.2 组合抽象

现在已经看到，open 将返回一个抽象而不是一个具体的值，我们需要确保这个抽象可以和其他类似函数组合在一起。设想一下要对某个账户执行的一系列操作，比如开户，然后在这个账户里做一系列的借、贷操作。这看起来像是在命令式编程中所做的一个序列的操作。但在函数化范式中，将给用户造成一种命令式顺序的感觉，但在这个错觉之下，做的其实是面向表达式的编程。而且在这里，函数所返回的抽象有很大的施展空间。

当我们有一个操作的序列时，如果所有操作都成功，那么每个操作的返回都需要排列好并生成最终的计算。如果任意一个操作发生了失败，则整个操作需要报告一个失败。在 Scala 中，通常会用 for 表达式来实现：

```
for {
  a <- open(..)
  b <- credit(..)
  c <- credit(..)
} yield(..)
```

图 3.2 展示了我们所讨论的组合是如何与计算结构一起工作的。这个抽象需要支持成功路径，同时也允许在遇到任何失败时能够终止操作的组合。

图 3.2 还展示了计算的单子（monadic）模型[1]，我们将在遇到序列操作调用时用它来链接 AccountService 的方法。每个服务函数的返回类型需要支持这个单子模型。这也是为了组合函数所采取的策略。通过这种方式，将操作链接起来，那么当我们说一个计算是一个 monad 时，就能明白它所代表的含义。在这里，计算只发生在一个抽象了某些语义的结构内部——对 AccountService 的 API 使用的 Try 就抽象了失败的语义。我们称这类计算是多作用的（effectful），而 monad 是组合多作用计算的一种方式。在第 4 章中讨论函数式编程的其他模式时，将学习到这类模型的更多内容。

1 这仅仅是“单子模型是什么”的一般描述——第4章中会更详细地讲解monad。

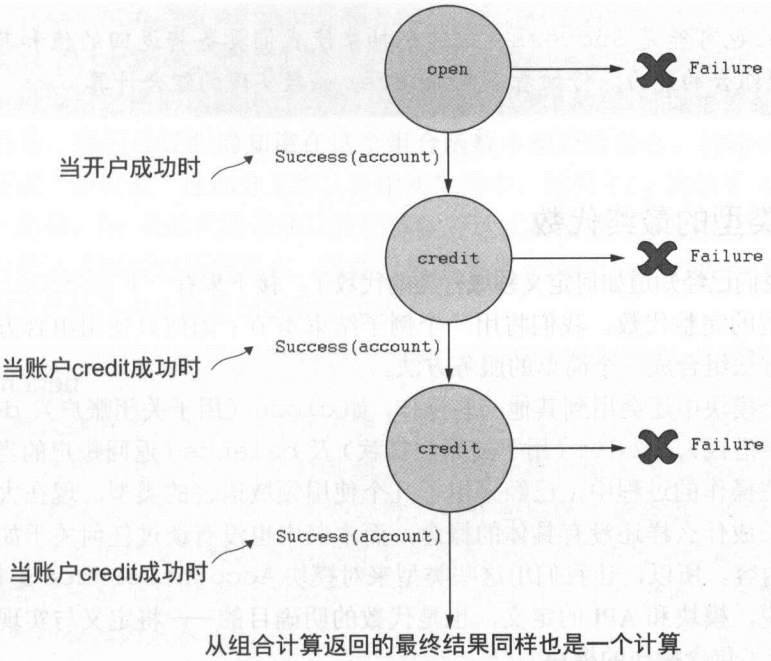


图 3.2 垂直的箭头描述的是成功路径。当一个操作成功时，将在计算的 Success 分支中获取结果，然后继续进入下一个操作。如果所有操作都成功了，会将最终计算作为结果。注意，这个链条是计算所体现出来的。一旦在任何阶段操作失败了，将进入计算的 Failure 分支。

在 Scala 标准库中，`Try` 被定义为一个 monad。[1] `Try` 实现了 `flatMap`，它能提供我们刚才所讨论的作用的排列。使用 `Try` 的组合性威力，可以通过组合一些原始的操作来定义出新的操作，马上我们就会看到如何用 `debit` 和 `credit` 来实现 `transfer` 操作。

这样，我们就掌握了一个如何用 monad 组合领域行为代数的技巧。对于所有特定的目的，一个 monad 就是一个计算的抽象，它可以支持一定数量的操作。在第 4 章中将详细介绍 monad 是如何作为一个通用计算结构用于领域模型设计的。

Try 作为一个抽象

使用 `Try[Account]` 而不是 `Account` 作为返回类型，对于 `open` 方法来说有什么好处呢？当返回一个账户（类型 `Account` 的一个实例）时，返回的是某个已经经过赋值的事物，同时这些值也没有组合。如果返回一个 `Try[Account]`，返回的是某个抽象了赋值结果的事物。这个结果可以是

1 Try违反了一些抽象设计的法则（参见https://issues.scala-lang.org/browse/SI-6284），因此它也不是一个合法的monad。在下一节中将学习更多有关代数的法则。第4章会详细讨论monad。

Failure 也可能是 Success，但这个抽象使我们具备将返回的值和其他抽象进行代数组合的能力。仔细看下 transfer 函数实现的组合计算。

3.2.3 类型的最终代数

现在我们已经知道如何定义领域行为的代数了，接下来看一下 AccountService 发布的类型的完整代数。我们将用一个例子结束本节：如何只使用组合方法的代数将一些小方法组合成一个简单的服务方法。

在这个模块中还会用到其他一些操作，如 close（用于关闭账户）、debit（用于从账户中借钱）、credit（用于给账户贷款）及 balance（返回账户的当前余额）。在定义这些操作的过程中，已经采用了几个使用领域语言的类型。现在大家可能对这些类型长成什么样还没有具体的概念，而本书中也没有谈过任何关于如何实现这些类型的内容。所以，让我们用这些类型来对模块 AccountService 进行参数化。如之前所说，模块和 API 的定义，也是代数的明确目的——将定义与实现解耦。清单 3.1 展示了包含操作的模块。

清单 3.1 包含操作的 AccountService 模块

```
trait AccountService[Account, Amount, Balance] {
  def open(no: String, name: String, openDate: Option[Date]): Try[Account]
  def close(account: Account, closeDate: Option[Date]): Try[Account]
  def debit(account: Account, amount: Amount): Try[Account]
  def credit(account: Account, amount: Amount): Try[Account]
  def balance(account: Account): Try[Balance]
}
```

类型参数化

每个函数都返回一个 Try。前文提过，这样做有两个原因。首先，有助于向用户传递失败信息。其次，有助于序列化操作，因为 Try 是一个 monad。这里有个例子，如何用 debit 和 credit 组合一个新的函数 transfer：

```
def transfer(from: Account, to: Account, amount: Amount):
  Try[(Account, Account, Amount)] = for {
    a <- debit(from, amount)
    b <- credit(to, amount)
  } yield (a, b, amount)
```

仅仅利用代数本身，我们们已经定义了一个业务操作的完整实现。函数 transfer 的代数由 3 个类型组成：获取的类型、返回的类型以及定义领域行为的业务规则的类型。要注意的是，在谈论代数时，说的只是组成规则的类型，而不是

实现。我们完全不知道类型 Account 和 Balance 是什么样的，也不知道将如何实现行为 debit 或 credit，但我们有一个用它们组成的函数的完整实现。这就是代数的组合：可以对之前的函数进行组合，只是因为代数的类型可以很好地匹配。还要注意到的是，我们已经明确知道在这个组合函数中想要做什么。神奇的是，Try 数据类型完成“如何做”这部分工作。在现实生活中，返回 Try 的单子 API 是和操作绑定在一起的。for 表达式的语法让我们感觉表达式都被按顺序地执行，但内部却保留了面向表达式编程的所有优点。现在已经需要将 transfer 函数包含进模块定义中，因为它已经不再依赖任何函数的实现。

什么是 monad

在函数式编程中，我们将计算声明为表达式。一个表达式既可以是一个原始的计算，也可以是由多个简单计算组合在一起并得出结果的复杂计算。我们将领域行为建设成领域模型的组成部分时，同样也实现了组合器，它也是用类似的方式进行进化和演变的。我们从简单函数开始，然后运用高阶函数的威力，将它们组合在一起来设计更大的抽象。一个 monad 抽象了计算的一个类型，用于建立这种组合器的基本库。

一个 monad 由以下 3 个元素组成：

- 一个类型构造器 M[A]，Scala 中一般表述为 trait M[A]、case class M[A] 或者 class M[A]。
- 一个 unit 方法，它将一个计算引入 monad。在 Scala 中，使用类构造器的调用来达到这个目的。
- 一个 bind 方法，它将计算序列化。在 Scala 中，flatMap 就是这个 bind。

这是在暗示 monad 是一个代数结构。任何拥有这 3 个元素的 monad 同样要遵守以下 3 个原则。

- 一致性（identity）：对一个单子 m，m flatMap unit => m
- 单元性（unit）：对一个单子 m，unit(v) flatMap f => f(v)
- 组合性（associativity）：对一个单子 m，m flatMap g flatMap h => m flatMap{x=>g(x) flatMap h}。

3.2.4 代数法则

前面提过，迈向基于代数的 API 设计的其中一个步骤就是形成代数的法则。我们需要明确规定一些 API 必须遵守的不变式。这些将成为通用约束，或者它们也可以被领域中的规则所驱动。

让我们看一个例子。下面是一个必须强制遵守的基本法则：对所有的账户，给

定一个余额 B，执行一次相同数目的 credit 和 debit，成功后将返回 B。用 API 定义，可以规定这个法则作为模型的属性，并记录为测试用例的一个部分。在第 9 章中讨论基于属性的测试时，将学到如何检验模型的不变式。下面是该法则的一个例子：

```
property("Equal credit & debit in sequence retain the same balance") =
  forAll((a: Account, m: BigDecimal) => {
    val Success((before, after)) = for {
      b <- balance(a)
      c <- credit(a, m)
      d <- debit(c, m)
    } yield (b, d.balance)

    before == after
  })
```

这个代码片段使用了一个名字为 ScalaCheck 的库（http://scalacheck.org），它在 Scala 中是一个用于基于属性测试的通用库。在第 9 章中将学习这个工具。这个讨论的结果是通过使用这种不变式并将它们编码成可检验的属性，不仅仅可以规定领域模型的强制约束，同时也可以在每次构建系统时执行它们——这也是基于代数的 API 设计的另外一个重要方面。

法则使得聚合保持一致

聚合最重要的属性就是它定义了抽象的一致性边界。不管在聚合中做什么样的操作，它一定不会与模型的观点不一致。代数法则必须确保这一点。大部分规则将被类型系统所检验，而另一部分将使用基于属性的测试来检查，这在 3.2.4 节一开始就已经讨论过。

在清单 3.1 的模块案例中，其中一个一致性保证就是当关闭一个账户时，账户的关闭日期必须大于开户日期——这也意味着模块 `AccountService` 实现的任何对 `Account` 的操作都不能违背这个不变式。很明显，最有可能这么做的函数就是 `close` 操作。因此，作为该模块必须遵守的法则，必须在账户聚合上强制执行它。在下一节中将看到当实现模块以及 `close` 操作时如何强制执行该法则。

练习 3.1 对失败建模的多种方式

在本节中，用 `Try` 作为对失败建模的计算结构。让我们看一下其他的选项，并找出这些途径的相关优点：

- 用 `Option` 作为方法的返回类型重写 `AccountService` 的定义。提示：`Option` 同样实现了 `flatMap`，而且也可以用于链式计算。大家会不会认为 `Option` 在建模中会是比 `Try` 更好的一种方式呢？
- 用 `Either` 来对操作的失败建模。跟 `Try` 一样，`Either` 也是一个和类型，

可以用来切分成功和失败分支。那么在链接序列中的操作时，它会不会是一个比 Try 更适合的选择？

3.2.5 代数解释程序

前面章节中，代数总结了领域模型所要遵守的规定。大家现在可能在想，API 的实现会是什么样的呢？我们的思想是将实现与代数本身进行解耦，这样单一代数就可以拥有多个实现。每个实现被认为是代数的解释程序，并组成具体的类和函数，实现 API 的定义。

在清单 3.1 中，当定义 API 时，提炼了一些类型（比如 Amount，Account，Balance），并用它们参数化了 AccountService。如清单 3.2 所示，现在是时候为每个类型提供具体实现了。针对清单 3.1 中定义的代数，清单 3.3 中还提供了一个解释程序的例子。

清单 3.2 AccountService API 实现的基本抽象

```
type Amount = BigDecimal
def today = Calendar.getInstance.getTime
case class Balance(amount: Amount = 0)
case class Account(no: String, name: String, dateOfOpen: Date,
  dateOfClose: Option[Date] = None, balance: Balance = Balance(0))
```

清单 3.3 代数解释程序

```
object AccountService extends AccountService[Account, Amount, Balance] {

  def open(no: String, name: String, openingDate: Option[Date])
➥ : Try[Account] = {
    if (no.isEmpty || name.isEmpty)
      Failure(new Exception(s"Account no or name cannot be blank") )
    else if (openingDate.getOrElse(today) before today)
      Failure(new Exception(s"Cannot open account in the past"))
    else Success(Account(no, name, openingDate.getOrElse(today)))
  }

  def close(account: Account, closeDate: Option[Date]): Try[Account] = {
    val cd = closeDate.getOrElse(today)
    if (cd before account.dateOfOpen)
      Failure(new Exception(s"Close date $cd cannot be before
➥ opening date ${account.dateOfOpen}"))
    else Success(account.copy(dateOfClose = Some(cd)))
  }

  def debit(a: Account, amount: Amount): Try[Account] = {
    if (a.balance.amount < amount)
      Failure(new Exception("Insufficient balance"))
```

```
    else Success(a.copy(balance = Balance(a.balance.amount - amount)))
  }

  def credit(a: Account, amount: Amount): Try[Account] =
    Success(a.copy(balance = Balance(a.balance.amount + amount)))

  def balance(account: Account): Try[Balance] = Success(account.balance)
}
```

图 3.3 描述如何拆分代数和解释程序。对单一代数来说，我们也可以拥有多个解释程序。如需进一步了解所有的实现细节以及完整的可运行代码，请参考本书的代码下载资源。

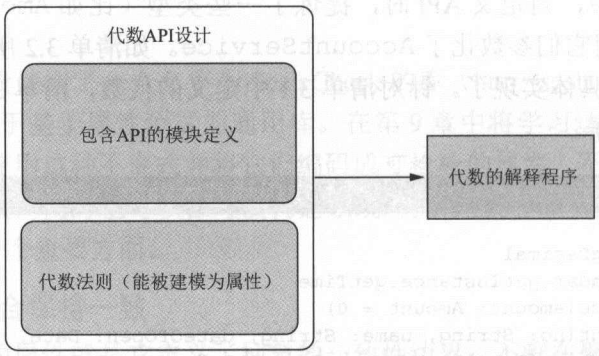


图 3.3 基于代数的 API 设计。模块包含 API 定义，而且通常用 Scala 的 trait 来实现。法则通常用属性来实现，而且可以用明确的方式来校验。代数的解释程序就是 API 的实现，而且是从定义中被解耦出来的。

3.3 领域模型生命周期中的模式

第 1 章中提供了以下 3 个阶段来组成领域对象的生命周期。

- 创建：通过组件创建对象。
- 参与：在领域中，一个合法的领域对象与其他抽象交互来传递函数性。
- 持久化：领域对象被写入某种持久化存储中。

任何时候，不管用何种实现技术，谈到领域模型的实现时，这些都是最普遍的模式。接下来针对个人银行业务领域中最重要的对象——客户账户，分析一下这 3 个生命周期阶段。任何时候去最近的银行，跟慈眉善目的银行柜员要求开一个账户，银行后台系统都会为我们“创建”一个 Account 对象。作为账户的保有者，它必须是一个保存有我们详细信息的合法对象，然后系统会给它分配一个合法的账户编号作为唯一身份证明标签。账户一旦被创建，就开始参与各种与账户相关的交易（比如借贷和转账），这将直接影响账户的状态和余额。同时还会参与到各种结算类的

操作，比如在账户中存入利息。所有这些都是“参与”阶段的例子，在这里对于银行活动来说领域对象是固有的组成部分。账户详细信息不可能一直储存在系统的内存中——有时需要将账户以及交易保留在一个存储中，同时还能提供查询、更新以及删除的能力。与所有领域对象聚合有关的这些生命周期模式构成了系统的完整模型。

本章涵盖了其中一些模式的实现，会用到函数式编程中一些通用的技术，以及Scala中一些特有的技术。我们可以用多种方式来创建对象——最简单的方式就是直接调用类构造器。但是，最简单的技术通常会有一些陷阱，而且在绝大部分非传统的使用场景中它看起来会很傻很天真。在为领域模型设计任何创建型的模式时，需要运用恰当的软件工程原则。创建的过程同样需要被验证——我们所创建的对象必须满足最低程度的有效性，后文会进一步解释。

在对象通过了领域确认阶段，即可得到一个完整创建的有效的领域聚合。就像在第1章中所学的，一个聚合是个完整的领域实体，它描述了领域模型的核心概念。第1章中关于`Account`的例子毫无疑问是模型中最重要的主角之一。（如果需要回顾领域聚合，可以查看第1章里更多的细节。）确认是一个非常重要的话题——确认必须对对象可插拔，而且在不同的约束下可重用，它们还需要具有特定的失败语义和领域友好的语法。接下来将看到函数组合是如何为所有这些情况提供一个完美的方案的。稍后将在本章中会讨论*领域确认模式*。

现在已经有了一个聚合（一个合法的领域实体），大家一定在想是什么组成了聚合的内部实现。基于类的OO技术教会了我们如何将数据元素和函数捆绑在相同的模块化结构中，最终得到了一个看起来很丰满的领域元素层次。OO的追随者称其为*富领域模型*，而“丰满”程度直接取决于相同类层次下所能捆绑的结构和函数的数目。在函数式设计中，讨论的是“瘦”领域对象——只打包必须的结构来描述抽象的核心概念。我们将使用代数数据类型来建模瘦元素。通过可重用可扩展的结构，也称为*模块*，函数将具备分布式特性，这也形成了领域的代数。每个模块大致代表了一个业务功能单元——比如，一个`Portfolio`模块可能要处理不同的功能，包括计算和报告客户的投资组合。这种模块化的方式使得大型模型更容易被理解，它的构建方式对用户来说更加直观。而且，如果使用的语言支持头等模块，可以用小型模块组合成大型模块，同时促进模型功能的有机进化。这样反过来也可以使模型有更好的可重用性和可维护性。

一个聚合将参与不同的领域行为。我们将学到如何在抽象方面对这种行为模式进行建模，这样就可以更加高效地重用，并且可以经常用数学的方式证明它们的正确性。大部分抽象都是合法的：它们将遵守特定的法则，防止它们在领域生命周期的任何阶段发生不稳定甚至错误。我们要保证它们的正确性，并运用类型系统的力量来确保正确。

我们还将看到持久化模式——主要是从领域的主线工作流中解耦持久化。不用关注特定的数据库技术，但要确保实现在与多种持久化存储交互时能够足够地通用。

让我们从一些领域对象创建的相关模式开始，看一下如何给用户提供一个统一的接口来创建相关模型元素的家族。

3.3.1 工厂——对象从何处来

如果在使用基于类的语言，可以通过调用类构造器来创建对象。Scala 是一种基于类的语言，但我们将用它的函数抽象来做创建——代数数据类型，就如 case class 所实现的。前面的章节中已经有过用 case class 构建对象的例子。在本节中，将结合更大的领域模型进一步丰富创建的语义。

模型越复杂，就越要更好地管理抽象。这也是为领域元素采用特殊创建策略的驱动力之一。工厂为创建对象提供了一个标准接口，但是术语 *factory* 只是模式词汇表的一个组成部分而已——它是这个模式的实现，在不同的编程语言和范式中可能都会不同。Scala 提供 case class 作为对不变领域元素的建模方式，而 case class 免费提供一个伴侣对象，用于创建的默认工厂。可以用它们来创建领域元素。但当我们讨论复杂模型的部署和管理时，仍有很多其他关于对象创建的问题需要考虑：

- 如何保证工厂返回一个有效的对象给客户？
- 在哪里开始确认逻辑？
- 如果确认失败，将会发生什么——会抛出异常吗？但如第 1 章中所说，异常违背了应用透明。

在对象构建上必须保持足够的聪明。工厂需要返回一个最小可用的对象，这也就意味着它不能有不合法或不稳定的核心成分。可能是某些具体的业务确认，它们可能还没有被执行，但一些最基本的诸如有一个负数的年龄字段是绝对不被允许的。

3.3.2 智能构造器

易于构建对象的标准技术需要遵守一系列的约束，这也是通常所说的智能构造器。我们禁止用户调用代数数据类型的基础构造器，并提供一个更聪明的版本确保用户能获取一个数据类型，从中用户不仅能获得一个有效的领域对象实例，还能获取失败的相关描述。让我们来看一个例子。

在个人银行领域中，很多工作需要安排在每周特定的某些天来执行。这里就产生了一个抽象——可以实现“每周的某一天”，这样就可以在构造的过程中对其进行校验。将每周的某天描述为一个整型值，但它需要遵守一些约束，才能成为合法的“每周的某一天”——它的值必须介于 1 到 7 之间，1 表示周一，7 表示周日。大家会不

会像下面这样去实现？

```
case class DayOfWeek(day: Int) {
  if (day < 1 or day > 7)
    throw new IllegalArgumentException("Must lie between 1 and 7")
}
```

这就违反了我们的基本准则，应用透明——异常不是函数式编程里的友善公民。让我们采用一个更聪明的方法。清单 3.4 展现了该抽象的智能构造器。先看代码，再来剖析它这样做的理由。

清单 3.4 使用智能构造器的 DayOfWeek

```
sealed trait DayOfWeek {                          <-- 用于每周某天建模的核心抽象
  val value: Int

  override def toString = value match {
      case 1 => "Monday"
      case 2 => "Tuesday"
      case 3 => "Wednesday"
      case 4 => "Thursday"
      case 5 => "Friday"
      case 6 => "Saturday"
      case 7 => "Sunday"
  }
}                                                 <-- 模块的伴侣对象

object DayOfWeek {
  private def unsafeDayOfWeek(d: Int) = new DayOfWeek { val value = d }
  private val isValid: Int => Boolean = { i => i >= 1 && i <= 7 }
  def dayOfWeek(d: Int): Option[DayOfWeek] = if (isValid(d))
    Some(unsafeDayOfWeek(d)) else None            <-- 发布的智能构造器
}
```

接下来分析一下该实现所提供的特性是如何使其更加智能的：

- 用来创建 DayOfWeek 的主接口被命名为不安全的（unsafe），同时标记为私有的（private）。它不会暴露给用户，只能在实现的内部使用。用户无法通过调用该函数来获取 DayOfWeek 的实例。我们是有意这样做的，因为如果用户将一个超出范围的整型值作为参数传入该函数，得到的实例就不是合法的。
- 获取数据类型实例的唯一手段就是使用伴侣对象 DayOfWeek 的智能构造器 dayOfWeek，不管构造对象最终是否成功。
- 留意智能构造器的返回类型，它是 Option[DayOfWeek]。如果用户传入一个合法的整型值，将取得一个 Some(DayOfWeek)，否则就会得到 None，代表没有值。

- 为了使案例尽量简单，Option 被用来体现实例构造的可选状况。但对于可能有多种复杂校验逻辑的数据类型来说，客户可能需要知道对象创建失败的具体原因。这可以使用更有表现力的数据类型比如 Either 或者 Try 来实现，它们会允许返回创建失败的原因。在下一个案例中将会看到这种用法。
- 大部分创建和校验的领域逻辑都从核心抽象（trait）移到了伴侣对象（一个模块）中。与 OO 所支持的富模型相反，这就是之前所说的瘦模型实现。
- 智能构造器的典型调用是 DayOfWeek.dayOfWeek(n).foreach(schedule)，在这里 schedule 是个函数，它为所获取的 DayOfWeek 安排一个工作。

最后用领域中相关对象的创建作为例子来结束本节。我们可以在创建函数中使用更有表现力的类型，这样创建过程失败时，它将返回详细的失败信息给用户。

什么是封闭的 trait

在 Scala 中，如果所有 trait 或 class 的子类型都在同一个文件里，那么该 trait 或 class 就是封闭的，封闭 trait 也是在这里被定义的。

如果将 trait 变成封闭的，编译器需要知道所有可能拥有的子类型，这样它才能推导。如果在模式匹配中使用 trait，但又没有在匹配中包含所有子类型，编译器就将抛出警告并提示匹配不彻底。

在为领域元素使用封闭 trait 之前要先修复子类型的数目问题。

3.3.3 通过更有表现力的类型进一步提升智能

我们可以创建不同类型的账户——比如支票和储蓄账户——同时创建和确认逻辑依然通过模块系统内部的智能构造器来进行。仔细看一下清单 3.5，就会发现它与清单 3.4 的实现有些细微的不同。最显著的不同就是创建函数返回了一个更有表现力的类型（Try）。如之前所说，如果客户端代码需要返回失败信息给上游的话，Try 将返回构造失败的原因。通过强有力的类型系统，只需要让模型实现具有足够的表现力就可以了，一点都不会影响编译时的安全。

清单 3.5 使用智能构造器创建 Account

```
import java.util.{ Date, Calendar }
import util.{ Try, Success, Failure }

type Amount = BigDecimal
def today = Calendar.getInstance.getTime
```

```
case class Balance(amount: Amount = 0)

sealed trait Account {
  def no: String
  def name: String
  def dateOfOpen: Option[Date]
  def dateOfClose: Option[Date]
  def balance: Balance
}
final case class CheckingAccount private (no: String, name: String,
  dateOfOpen: Option[Date], dateOfClose: Option[Date]
  balance: Balance) extends Account

final case class SavingsAccount private (no: String, name: String,
  rateOfInterest: Amount, dateOfOpen: Option[Date],
  dateOfClose: Option[Date], balance: Balance)
  extends Account

object Account {
  def checkingAccount(no: String, name: String, openDate: Option[Date],
    closeDate: Option[Date], balance: Balance): Try[Account] = {

    closeDateCheck(openDate, closeDate).map { d =>
      CheckingAccount(no, name, Some(d._1), d._2, balance)
    }
  }

  def savingsAccount(no: String, name: String, rate: BigDecimal,
    openDate: Option[Date], closeDate: Option[Date],
    balance: Balance): Try[Account] = {

    closeDateCheck(openDate, closeDate).map { d =>
      if (rate <= BigDecimal(0))
        throw new Exception(s"Interest rate $rate must be > 0")
      else
        SavingsAccount(no, name, rate, Some(d._1), d._2, balance)
    }
  }

  private def closeDateCheck(openDate: Option[Date],
    closeDate: Option[Date]): Try[(Date, Option[Date])] = {
    val od = openDate.getOrElse(today)

    closeDate.map { cd =>
      if (cd before od) Failure(new Exception(
➥ s"Close date [$cd] cannot be earlier than open date [$od]"))
      else Success((od, Some(cd)))
    }.getOrElse {
      Success((od, closeDate))
    }
  }
}
```

Account 的基本内容

将支票账户和储蓄账户定制实现为 ADT

创建账户的智能构造器

前面结合个人银行领域的两个具体例子非常详细地讨论了智能构造器。现在我们已经十分了解该技术在何种使用场景下会显得非常有价值。下面汇总了几个非常有用的结论：

- 当所要实例化的领域对象需要遵守一系列的约束时，智能构造器是非常管用的。
- 防止用户接触类构造器，因为直接使用它可能会导致不安全且不确定的创建。类声明中的 private 能防止 case class 的直接实例化，而关键字 final 可以阻止继承。
- 发布的 API 对于失败场景必须做到非常明确，同时返回给用户的数据类型必须具备足够的表现力。
- 对于不需要明确校验的简单对象，可以直接使用类构造器。使用合理命名的参数，使得构造器的调用足够清晰。

本节涉及在领域对象的生命周期中经常遇到的通用模式，还有创建模式以及实现的例子。在后续的章节中将会有其他的模式，诸如聚合模式和仓储模式。还将了解到它们的实现与在 OO 编程中遇到的类似的结构有什么不同。最好重新使用函数式编程，这样将看到如何用函数技术实现聚合和仓储，同时使它们可重用、可扩展并易于测试。

3.3.4 用代数数据类型聚合

正如前面所讨论的，聚合为外部世界访问实体提供了一个单独的引用入口。把 Account 看作一个实体时，Account 可能包含其他实体，比如 Address[1]，或其他值对象，比如日期。但作为 API 的一个客户端，可能更希望操作账户时不需要关注组成 Account 其他元素的实现细节。

让我们想一下如何实现一个客户的投资组合。一个投资组合由一系列仓位的集合组成，它表明一个客户在一个特定日期所持有的不同货币余额。比如说，客户 John Doe 在 2014 年 3 月 26 日有如下投资组合：

- 账户 E123 有余额 2300 欧元。
- 账户 U345 有余额 12000 美元。
- 账户 A754 有余额 3456 澳元。

如何将这个投资组合建模为一个聚合，使 API 的客户不需要处理 Position 或 Money 这样不同的独立元素？以下是实现聚合的一种方法：

```
sealed trait Account
sealed trait Currency
case class Money(amount: BigDecimal)
case class Position(account: Account, ccy: Currency, balance: Money)
case class Portfolio(pos: Seq[Position], asOf: Date)
```

1 第1章中讨论过，Address既可以是一个实体，也可以是一个值对象，这取决于边界上下文。

这里我们拥有了一个关于投资组合的聚合，它用 ADT Portfolio 作为聚合根。还有很多其他元素组成了聚合的完整结构，包括实体如 Account，值对象如 Date、Money 及 Currency。对客户来说，Portfolio 是最重要的，也是交互的唯一接口，可以通过封装一个用户投资组合实现的所有细节来确保这一点。通过聚合根发布方法，可以只获取用户有限的相关细节并反馈适当的信息：

```
case class Portfolio(pos: Seq[Position], asOf: Date) {
  def balance(a: Account, ccy: Currency): BigDecimal =
    pos.find(p => p.account == a && p.ccy == ccy)
       .map(_.balance.amount).getOrElse(0)

  def balance(ccy: Currency): BigDecimal =
    pos.filter(_.ccy == ccy)
       .map(_.balance.amount).foldLeft(BigDecimal(0))(_ + _)
}
```

获取一个特定账户特定货币的余额

获取所有账户特定货币的余额

在这里设计了一个稍微有点胖的聚合——它包含多个实体，比如 Portfolio（聚合根）和 Account，账户指明生成谁的投资组合。在很多情况下这种设计可能不好衡量，特别是面对一个大型模型时。有多个实体参与一个聚合时，要对不变式和处理强制限制上下文就会变得非常困难。这时通常可以对其进行重构，使 Position 只包含账户编号而不是整个 Account 实体。因为一个账户编号对应唯一一个账户，实现可以在需要时再构建一个 Account 实例。这样聚合就只有一个实体 Portfolio（聚合根），而其他均为值对象。[1]

在典型的 OO 实现中，我们将操作 Portfolio 的相关函数包含在类自身的内部。但在本节前面也提到了，对于函数式编程来说，需要将抽象保持简约瘦小，所以 Portfolio ADT 只会包含组成投资组合最少的结构。所有相关函数都被移到提供函数定义的模块中，而用户可以使用该模块。回到 3.1 节可以回顾如何设计模块的代数来组织领域函数。

本次讨论的结论就是聚合由代数数据类型（a）组成，这些类型组成了实体的结构，而模块（b）提供了用组合方式操作聚合的代数。

模块化

拆分一个聚合的两个部分对于设计领域模型是一个非常重要的概念。这会使设计变得模块化，同时使聚合的结构和函数有一个清晰的划分。聚合的实现结构一定不能泄露给客户端，哪怕是某些函数式编程技术，比如模式匹配，因为这些可能会使实现细节暴露给用户。除非有非常重要的原因，同时也要确保所有聚合的操作都通过合法的模块代数来完成。

1 了解更多聚合设计准则，请参考VaughnVernon所著的*Effective Aggregate Design*，2011年（http://dddcommunity.org/library/vernon_2011/）。

关于模式匹配的警告

在 Scala 中，结合 case class 与模式匹配的使用频率是非常高的。对用户来说，它的语法非常便利，而且有足够的表现力和可读性。但作为领域建模者，模式匹配有两个弱点必须搞清楚：

- 模式暴露了对象表现，也因此被认为是反模块的。
- 不能为了匹配而去定义用户模式——它们是和 case class 的类型一对一捆绑好的。因此，模式是不可扩展的。

在很多情况下，可能要考虑使用 Scala 的提取器（extractor）模式。关于为何提取器是模式匹配的升级，更多的信息请参考 Burak Emir、Martin Odersky 和 John Williams 所著的论文。[1]

不变式

在讨论一个模型以及它相关的元素时，所有的函数都是基于领域规则的。在任何情况下都不可以违反这些规则。比如说，不能创建两个有相同编号的账户，也不能使一个账户的关闭日期早于开户日期。

当创建聚合时，确保这类领域规则被遵守是 API 的职责。可能有些复杂的业务规则需要在创建聚合之后才能被校验，但新建对象必须通过基本的校验，而领域必须决定哪些规则是“基本的”。在 3.2.2 节中讨论的智能构造器是确保创建时不变式能被有效遵守的一种技术手段。

3.3.5 用透镜更新聚合功能

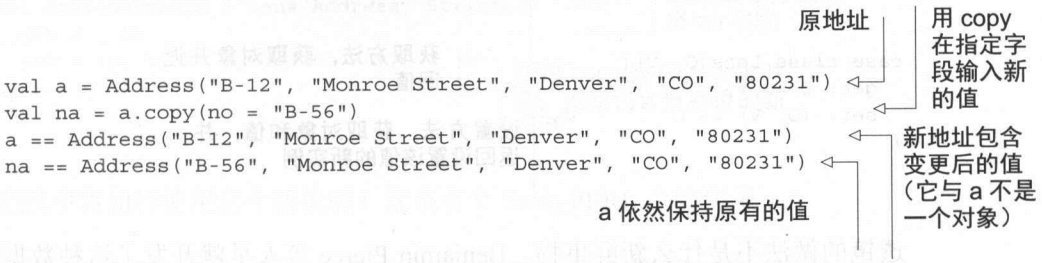
到目前为止，我们一直聚焦在不变性和引用透明上——现在来讨论一下更新。这看起来是不是有点矛盾呢？事实上，更新是一种自然现象，我们所设计的每个领域模型都必须提供领域元素的更新能力。

在 Scala 中，可以用 `var` 来处理对象的不受限变更。不过因为显而易见的原因，我们不想掉进这个陷阱里——在函数式编程范式中，它不会带来有意义的帮助。在函数式编程中最常见的做法是避免空间内的变更，用需要更新的值生成对象的一个新的实例。Scala 的 case class 为这个操作提供了句法上的小糖块。这里有一个例子：

```
case class Address(no: String, street: String, city: String,
  state: String, zip: String)
```

1 Burak Emir等人所著的*Matching Objects with Patterns*（http://lampwww.epfl.ch/~emir/written/MatchingObjectsWithPatterns-TR.pdf）。

在模型中，ADT（Address）对一个用户的地址进行了建模。如果想要更新一个属性（比如说，地址中房子的 no），用 Scala 的句法来做是非常直观的：




```
val a = Address("B-12", "Monroe Street", "Denver", "CO", "80231")
val na = a.copy(no = "B-56")
a == Address("B-12", "Monroe Street", "Denver", "CO", "80231")
na == Address("B-56", "Monroe Street", "Denver", "CO", "80231")
```

这看起来很酷，而且也没有违反不变性的原则。但让我们看一下如果这个方式发生了变化会有什么影响。引入我们的 Customer 实体并给它一个地址：

```
case class Customer(id: Int, name: String, address: Address)
```

同样使用这个 copy 方式来更新一个指定用户地址的 no 字段：

```
val c = Customer(12, "John D Cook", a)
val nc = c.copy(address = c.address.copy(no = "B152"))
```

能预感到会有什么问题发生吗？对象的嵌套层次变深了，句法也变得更加杂乱。想象一下，如果要更新一个深层嵌套的对象，copy 所要嵌套的层次会是什么样的。这种情况就需要更好的抽象——能保持不变性的优势，同时还能为用户提供一个好用的 API。那么就让我们转向设计抽象的代数途径。

在给出一个抽象的正式定义之前，先试着提出一些对代数的需求，也是这样一个抽象所要满足的内容。我们将这个抽象称为透镜（lens）。

- *参数化*：在需要更新的对象的类型上，透镜需要参数化（让我们称其为 O）。同时，因为每次更新都是针对对象的特定字段，那么针对需要更新的字段的类型透镜同样也需要参数化（让我们称其为 V）。这就提供了透镜的一个基本框架，`case class Lens[O, V] (..)`。
- *每个字段一个透镜*：对每一个对象的每一个字段，都需要有一个透镜。这听上去可能很冗余，而且在某些情况下它确实给人这样的感觉。稍后会看到如何使用基本库的宏指令来处理冗余。
- *获取方法*（getter）：透镜的代数需要为访问字段的当前值提供一个获取方法。它是一个很简单的函数，`get : O=>V`。
- *设置方法*（setter）：透镜的代数需要发布一个设置方法。它将获取一个对象和一个新的值，然后返回一个相同类型的新的对象，同时其字段的值变为新的值。很明显，它的函数是 `set: (O, V) => O`。

汇总这些点，就有了一个简单明了的透镜 ADT 实现，如下所示：

```
case class Lens[O, V](
  get: O => V,
  set: (O, V) => O
)
```

对象与值类型的参数化

获取方法，获取对象并返回值

设置方法，获取对象和值，并返回设置该值的新实例

透镜的做法不是什么新鲜事物。Benjamin Pierce 等人早就开发了这种数据结构用于解决双向转换的问题，诸如用相关设置[1]或树形结构数据[2]的转换来更新视图。如果对透镜作为计算抽象的历史感兴趣，可以参考以下文章。本节所讨论的实现是非常简化的。更加高大上的透镜实现包含 Scalaz（https://github.com/scalaz/scalaz）和 Shapless（http://github.com/milessabin/shapeless）。Gérard Huet’s Zipper（http://dl.acm.org/ citation.cfm?id=969872）是另外一种抽象，它能够对递归的数据结构做函数式更新。

让我们从个人银行领域中找个例子，看看如何用透镜实现更新。假设有一个 Address 的实例，它对银行用户的地址进行了建模，现在需要修改地址中房子的“no”属性。我们已经有了一个 Address 的 case class，就如之前所做的，它可以用 copy 函数很轻松地处理这个问题。但现在又有了一个亮晶晶的透镜抽象，想用它来做更新。这事现在看起来不是那么重要，但记住，我们的目的是提供一个好用的 API，用来嵌套更新用户对象地址里房子的 no 字段，所以就需要制定一个策略使它能够适应任意深度的嵌套对象。

让我们来定义一个透镜，在这里，对象类型是 Address，要更新的值（属性 no）是字符串类型：

1 这听上去可能比较复杂，关系型数据库的视图和这个视图的基础表在遇到更新的情况时，如何保持这两者之间的一致性，这里有可能会遇到一些问题。参见Aaron Bohannon，Benjamin C. Pierce与Jeffrey A. Vaughan所著的*Relational Lenses, a Language for Updatable Views*（http://dl.acm.org/citation.cfm?id=1142399）。

2 参见J. Nathan Foster，Michael B. Greenwald，Jonathan T. Moore，Benjamin C. Pierce与Allan Schmitt所著的*Combinators for Bidirectional Tree Transformations—A Linguistic Approach to the View Update Problem*（http://www.cis.upenn.edu/~bcpierce/papers/newlenses-full-toplas.pdf）。

```
val addressNoLens = Lens[Address, String](
  get = _.no,
  set = (o, v) => o.copy(no = v)
)
```

对象类型是 Address，属性 no 的类型是 String

通过获取方法取得 no 字段

使用 copy 函数的设置方法，返回包含更新值的新实例

实践中将如何使用这个透镜呢？这里有个 Scala PERL 上的例子：

```
scala> val a = Address(no = "B-12", street = "Monroe Street",
  city = "Denver", state = "CO", zip = "80231")
a: frdomain.ch3.lens.Address = Address(B-12,Monroe Street,Denver,CO,80231)

scala> addressNoLens.get(a)
res3: String = B-12

scala> addressNoLens.set(a, "B-56")
res4: frdomain.ch3.lens.Address = Address(B-56,Monroe Street,Denver,CO,80231)
```

获取函数返回当前 no 的值

设置函数返回另一个 Address 对象，它的 no 设置了新的值

类似地，可以定义一个透镜，用于更新 Customer 的 Address 字段：

```
val custAddressLens = Lens[Customer, Address](
  get = _.address,
  set = (o, v) => o.copy(address = v)
)
```

我们拥有的透镜可以和要更新的字段一样多。但 case class 中每个 copy 函数的嵌套问题还没有被解决。既然是真正的函数式编程，函数组合再次被引入我们的解决方案。函数要进行组合的前提是类型需要保持匹配。在之前的例子中，custAddressLens 将 Customer 映射到 Address，addressNoLens 将 Address 映射到 String。它们的类型也反映出相同的映射。现在看到匹配了吗？它就在那里等着被组合呢！但我们不会将这个组合实现成两个特定透镜的特殊用例，而是会定义一个通用的 compose 函数，这样就不需要为要组合的每一对透镜编写重复的代码。compose 将获取 2 个透镜和 1 个值，透镜的类型需要对齐以便于组合。接下来就只剩根据类型来定义获取函数和设置函数：

```
def compose[Outer, Inner, Value](
  outer: Lens[Outer, Inner],
  inner: Lens[Inner, Value]
) = Lens[Outer, Value](
  get = outer.get andThen inner.get,
  set = (obj, value) => outer.set(obj, inner.set(outer.get(obj), value))
)
```

需要组合的两个透镜，注意类型

组合器 andThen 组合两个 get 函数的调用

组合设置函数

现在可以用这个 compose 函数来创建一个更大的透镜，跨越 Customer 的嵌套数据，直接跳到更新用户地址内部的 no 属性。

```
scala> val a = Address(no = "B-12", street = "Monroe Street",
  city = "Denver", state = "CO", zip = "80231")
a: frdomain.ch3.lens.Address = Address(B-12,Monroe Street,Denver,CO,80231)

scala> val c = Customer(12, "John D Cook", a)
c: frdomain.ch3.lens.Customer = Customer(12,John D Cook,
➥ Address(B-12,Monroe Street,Denver,CO,80231))

scala> val custAddrNoLens = compose(custAddressLens, addressNoLens)
custAddrNoLens: frdomain.ch3.lens.Lens[frdomain.ch3.lens.Customer,
➥ String] = Lens(<function1>,<function2>)

scala> custAddrNoLens.get(c)
res0: String = B-12

scala> custAddrNoLens.set(c, "B675")
res1: frdomain.ch3.lens.Customer = Customer(12,John D
    Cook,Address(B675,Monroe Street,Denver,CO,80231))
```

之前定义的 Address 和 Customer

类型的组合透镜 [Customer, String]

组合透镜的获取函数，返回“B-12”

组合透镜的设置函数，它将返回一个新的 Customer，地址中的 no 被设置为“B675”

在领域模型中，聚合的一个目的就是控制通过聚合根访问底层实现。API 的用户不能直接访问低阶非根聚合元素，但可以通过用透镜组合简便地实现。暴露允许仅通过根元素进行低阶对象转换的顶层透镜。

透镜的使用

关于透镜的一个常见问题就是决定什么时候使用它们。Scala 里 case class 的 copy 函数难道不能满足处理函数更新的需要吗？就像许多其他抽象一样，透镜是可伸缩的。对于一个 ADT 内部的简单更新，copy 函数可以良好地工作，而且可能也是我们比较推荐的方式。但要考虑到下面的场景：

- 需要在深层嵌套的 ADT 中执行更新。在这种情况下，copy 就会变得难以驾驭，此时应该使用透镜。

- 需要和其他抽象组合 ADT 的更新。一个常见的例子就是和 State 单子进行组合。State 是管理应用状态的一个方式，它不需要使用空间内变更就可以随时改变。在使用单子 State 时，我们可以有效利用透镜来做此类更新。在第 4 章中会有相关的案例。

以上是透镜的简单介绍。本书的在线代码库中包含用于领域实体 Customer 的透镜完整实现，可以通过它学习透镜是如何适应代码整体架构的。观察非传统领域的透镜实现，可能会认为为实体的每个属性都要单独定义透镜会过于冗余，也会担心对于解决的问题来说实现中的这种冗余是否值得。首先，可以使用合适的库来解决冗余问题，它们使用 Scala 宏提供了透镜的免费样板实现。Monocle（https://github.com/julien-truffaut/Monocle）就是一个这样的库。Scalaz 和 Shapeless 这两个也是很棒的库，它们实现了 Scala 中的函数式编程抽象并将透镜作为它们数据结构的一个组成部分来提供。

透镜法则

ADT 透镜定义了抽象的代数。回想一下，每个代数都配备了一套需要遵守的法则来确保它的一致性。一般来说，每个透镜都需要遵守 3 个法则——它们看起来有点琐碎，但很值得将它们记录下来，而且只要我们定义了一个新的透镜，都要用基于属性的测试来校验它们：

- 一致性：如果获取并用相同的值设置回去，对象应该保持不变。
- 保存性：如果设置了一个值，紧接着又执行了一次获取，将取得所设置的值。
- 两次设置：如果连续做了两次设置并接着执行一次获取，将取到最后一次设置的值。

在领域模型中，当定义透镜来更新聚合时，不要忘了确认透镜需要满足的法则。毕竟聚合定义了模型的一致性边界。这也是作为设计师的职责，要确保所有逻辑的不变式都能被聚合中执行的所有操作所遵守。

一旦学会了如何使用基于属性的测试，就可以为实现的任意透镜校验属性。一种好的实践通常会在代码中明确包含这些法则——毕竟它们充当了可校验可执行的领域约束。

练习 3.2 验证透镜规律

第 3 章的代码库中包含一个 Customer 实体的定义以及它的透镜。观察 addressLens，它将更新一个用户的地址，然后用 ScalaCheck 写出属性来验证透镜规律。

3.3.6 仓储与解耦的永恒艺术

在第1章中介绍了仓储。仓储是聚合存在的地方，尽管组成形态可能有一些不同。但通常可以从仓储中构建一个聚合。具体如何做依赖于仓储的结构以及聚合和仓储实现范式之间的不匹配程度。仓储的另一个功能就是可以从仓储中查询聚合。在本节中，将学习如何做到以下几点：

- 用 Scala 的特性设计并实现一个仓储。
- 用 Scala 的 trait 在模块中组织仓储。
- 用组合的方式管理向领域服务中注入的仓储。

让我们从用于仓储模式的一个简单 API 开始。根据模型需求，可以在模型内部组织仓储。如果是很小型的模型，可以实现一个通用仓储，在单独一个模块中收拢所有聚合。通常在需要设计一个非传统的领域模型时，都需要为每个聚合设计独立的仓储，然后将它们组织成独立的 Scala 模块。比如我们当前的使用场景，需要处理 Account 聚合，同时要有一个专门的存储模块。仓储能够提供相关功能帮助我们完成以下内容：

- 根据 ID 匹配到一个聚合（记住，一个领域实体可以用一个 ID 唯一识别）。
- 存储一个完整实体。

如下，将这些函数抽象进一个泛型模块 Repository：

```
trait Repository[A, IdType] {
   def query(id: IdType): Try[Option[A]]
   def store(a: A): Try[A]
}
```

如果找得到的话，query 将会获取一个合法的 A，否则这里就会有一个异常。

然后要将这个泛型模块扩展成专门处理 Account 聚合的 AccountRepository：

```
trait AccountRepository extends Repository[Account, String] {
  def query(accountNo: String): Try[Option[Account]]
  def store(a: Account): Try[Account]
  def balance(accountNo: String): Try[Balance]
  def openedOn(date: Date): Try[Seq[Account]]
  //..
}
```

注意，函数的返回类型用 Try 来负责处理与仓储交互期间可能发生的失败。

在模块中组织仓储

有多个聚合时，可以把每个聚合的仓储组织成独立的模块。通过这个方法，可以获得清晰的代数划分以及随后的单元实现，它会将问题和解决方案领域建模成不同的实体。同时，因为 Scala 的模块可以被组合到一起，所以需要在一个

领域服务中同时使用它们时，可以将它们捆绑到一个独立模块中。这里有个例子：

```
trait PFRepos extends AccountRepository with CustomerRepository
with BankRepository
```

在模块中完成仓储的代数定义之后，就可以基于数据库完成具体实现，我们将用它来持久化仓储元素。以下是基于 Redis 键值存储的实现轮廓：

```
class AccountRepositoryRedis extends AccountRepository {
                                    实现略
}
```

往服务内注入存储

我们已经将仓储组织进不同的模块，也掌握了实现多个后端的方法，现在需要找到一个途径来把这些仓储注入一个领域服务。在领域模型中，服务是用户交互主要的粗粒度抽象。为服务制定一个良好的 API 是优秀领域设计的一个基本标准。所谓“好的 API”，也就是说它是简洁明了的，最重要的是，它是可以组合的。

一个领域服务需要访问仓储的 API——怎样建模这种交互呢？可以从最简单的开始——将仓储作为一个参数来传递。这是清单 3.1 中我们讨论领域服务的一个模块，它针对用户账户操作发布相应的 API[1]，在这里将仓储作为一个参数注入到服务方法中：

```
trait AccountService[Account, Amount, Balance] {
  def open(no: String, name: String, openingDate: Option[Date],
    r: AccountRepository): Try[Account]
  def close(no: String, closeDate: Option[Date],
    r: AccountRepository): Try[Account]
  def debit(no: String, amount: Amount,
    r: AccountRepository): Try[Account]
  def credit(no: String, amount: Amount,
    r: AccountRepository): Try[Account]
  def balance(no: String, r: AccountRepository): Try[Balance]
}
```

这是最简单的方法，但同时也是最天真的做法。考虑如下案例并试着找出其中的问题。这个例子是用 AccountService 发布的服务来计算用户账户的：

1 要注意与3.1节中的API代数有一个不同点。这里我们不会传递Account，传递的是一个账户编号，因为你需要使用仓储来查找账户聚合。

```
object App extends AccountService {
  def op(no: String, aRepo: AccountRepository) = for {
    _ <- credit(no, BigDecimal(100), aRepo)
    _ <- credit(no, BigDecimal(300), aRepo)
    _ <- debit(no, BigDecimal(160), aRepo)
    b <- balance(no, aRepo)
  } yield b
}
```

问题如下。

- 冗长：API 的用户需要将仓储作为一个强制参数传递给方法。这对于服务方法的单独调用是没有问题的，但如果组合了更大的抽象并进行多个序列的调用（就如前面这个例子），那它无疑是非常冗长的。
- 与 API 上下文捆绑在一起：对一个服务方法的每个调用，都要传递仓储的实体，这也就意味着仓储与 API 的上下文进行了强绑定。但在现实中，仓储形成了环境的上下文——它就像个存储，在那里被服务方法访问，并执行所要求的操作。
- 缺乏组合性：在这个实现中，将仓储以值的形式进行注入。通过计算上下文的方式来做注入，可以使 API 具有更好的组合性。其中的一个方式就是将参数提升为咖喱化形式。这也恰恰是我们接下来要做的。

想要组合性？咖喱化！

这三个问题的解决方案是使仓储成为一个咖喱化参数提供给服务方法。当通过 for 循环连续的调用时，可以通过计算传递仓储并推迟注入，直到组合函数的最后赋值。这种方式，既没有损失注入部分的任何内容，同时还获取了 API 的组合性优点。让我们结合用例看一个使用该技术的应用例子。首先，修改方法的代数来提升仓储参数到咖喱化形式（参见清单 3.6）。

清单 3.6 Function1 作为读取器（Reader）

```
trait AccountService[Account, Amount, Balance] {
  def open(no: String, name: String, openingDate: Option[Date]):
    AccountRepository => Try[Account]
  def close(no: String, closeDate: Option[Date]):
    AccountRepository => Try[Account]
  def debit(no: String, amount: Amount): AccountRepository => Try[Account]
  def credit(no: String, amount: Amount): AccountRepository => Try[Account]
  def balance(no: String): AccountRepository => Try[Balance]
}
```

所有方法返回 Function1

现在想想，这样写代码的时候会发生什么：

```
object App {
  import AccountService._
  def op(no: String) = for {
    _ <- credit(no, BigDecimal(100))
    _ <- credit(no, BigDecimal(300))
    _ <- debit(no, BigDecimal(160))
    b <- balance(no)
  } yield b
}
```

这段代码能够正确工作吗？如果给 Function1 一些额外的威力，也就是类型，就可以。相信大家现在已经意识到“额外的威力”就是 Function1 中定义的 flatMap 的威力。这不是由 Scala 标准库提供的，但可以使 Function1 成为一个 monad 并给它一个 flatMap。在第 4 章讨论 monad 时，将会学到更多细节。同时，如果大家有兴趣，第 3 章的在线代码库中有一个针对 Function1 的 monad 实现。

假设在 Function1 中已经定义了 flatMap，调用这个函数时，大家认为之前的计算 op 将返回什么？

```
scala> import App._
import App._

scala> op("a-123")
res0: AccountRepository => scala.util.Try[Balance] = <function1>
```

注意，复杂的表达式还没有被赋值——它只是获取返回的组合函数。现在需要将仓储明确传递给返回值来给整个计算做赋值。或者也可以将它与其他任意返回 Function1[AccountRepository,_] 的计算组合在一起并且推迟赋值，直到建立完整的计算管道。后一种方法也被称为“*通过增量组合建立抽象*”，是一种在函数式编程中作为规范的技术。这对于将仓储作为参数传递给 API 的方法来说是一个巨大的进步。

但如果想要将 op 和其他不是 Function1 的计算组合到一起呢？它可能是其他类型诸如 List 或 Option。当前的方法不能处理此类情况，因为缺少必要的黏合剂来组合多个单子作用。这是在下一节中要讨论的内容。

读取器（Reader）monad

有时候，函数所建模的计算除了明确传递的参数之外，还需要一些环境中附加的输入。读取器 monad 提供函数给我们去访问一个环境，从中可以读取所需要的信息，而不是让函数去访问全局的命名空间。刚才讨论的技术可以很好地完成此项工作。可以使用 AccountService 的一个 API 返回的函数来传递附加信息给计算。从实用的角度来说，如果需要一个读取器 monad 来访问仓储，都可以采用之前的实现技术。

在函数式编程中，组合建立计算管道并将计算推迟到结束位置是一种很常见的实践。前面章节中的计算 op 是个 Function1，它需要和包含 List 或 Option 的单子管道组合在一起。对于这种类型的计算将需要 monad 转换器。[1]但 Function1 并不具备任何可以用来将计算堆进一个单子管道的转换器。让我们通过另外一个层级的间接方式来解决这个问题。将 Function1 包装进另外一个抽象，这样就可以调用 Reader。在第 5 章中会看到，Reader 可以有一个转换器 ReaderT，它可以用来将其他 monad 和 Reader 进行组合。

```
case class Reader[R, A](run: R => A)
```

一个环境的应用 ADT，R，生成一个值，A

在这里，Reader[R, A] 只是一个函数的包装，它运行在环境 R 上并生成一个 A。在我们的使用场景中，R 就是仓储，而我们将从中读取一个 A。现在唯一要做的事就是实现 map 和 flatMap，使其能在管道中进行转换和读取的排序。清单 3.7 展示了 Reader monad，它已经准备好与领域服务 API 进行组合了。

清单 3.7 Reader monad

```
case class Reader[R, A](run: R => A) {
  def map[B](f: A => B): Reader[R, B] =
    Reader(r => f(run(r)))

  def flatMap[B](f: A => Reader[R, B]): Reader[R, B] =
    Reader(r => f(run(r)).run(r))
}
```

清单 3.8 展示了服务 API，其中更新了代数以体现这个改变。

清单 3.8 模块 API 与 Reader 的整合

```
trait AccountService[Account, Amount, Balance] {
  def open(no: String, name: String, openingDate: Option[Date]):
    Reader[AccountRepository, Try[Account]]
  def close(no: String, closeDate: Option[Date]):
    Reader[AccountRepository, Try[Account]]
  def debit(no: String, amount: Amount):
    Reader[AccountRepository, Try[Account]]
  def credit(no: String, amount: Amount):
    Reader[AccountRepository, Try[Account]]
  def balance(no: String): Reader[AccountRepository, Try[Balance]]
}
```

所有的 API 现在都返回一个 Reader 来读取 AccountRepository

1 第5章中将介绍monad转换器。

将 Reader 放在合适的地方，就是从用户视角所看到的服务 API 的组合，而且我们还没有传入任何仓储的具体实现。不管是使用一个确定的 Reader 抽象还是将 Function1 用作读取器，客户端代码都要保持相同：

```
object App extends AccountService {
  def op(no: String) = for {
    _ <- credit(no, BigDecimal(100))
    _ <- credit(no, BigDecimal(300))
    _ <- debit(no, BigDecimal(160))
    b <- balance(no)
  } yield b
}
```

这个代码片段好看而且实用，执行时将取得一个 Reader 实例。作为最后一步，调用 Reader 的 run 函数并传入环境参数，在我们的案例中就是 AccountRepository 的具体实现。运行 op("a123").run(AccountRepository)，它将在存款等一系列的操作之后返回给账户余额。使用这个组合策略，推迟赋值同样也会使测试变得更简单。因为将仓储的注入延缓到赋值阶段之前，所以可以为单元测试提供一个替换实现。如果仓储是基于一个企业级的后端数据库，就可以替换它，可以简单地插入一个内存中的实现来做服务的单元测试。本书的代码库中提供了一个使用读取器 monad 注入 AccountRepository 的 AccountService 的完整实现。

在这个案例中，读取器 monad 给我们带来如下好处：

- 防止实现交织在一起。[1]
- 通过减少应用代码中的样板文件，使实现更加领域友好。
- 将具体仓储实现的注入延迟到 API 的使用时（而不是定义时），这使设计更加模块化。该技术同样被称为依赖注入。[2]
- Function1 作为读取器，是 Reader monad 的具体实现。如果不需要和 monad 转换器组合，那可以放心地使用它。
- 因为读取器将计算从环境中解耦，就使得单元测试更加容易。替换实现并插入一个 mock，这样就可以很方便地开始单元测试了。

1 术语“去交织”（decomplect）首先被Rich Hickey用于*Simple Made Easy*一文中（www.infoq.com/presentations/Simple-Made-Easy/）。

2 通过一个强有力的语言，可以不需要任何框架来做依赖注入。语言本身就提供了所有的机制。这里的Reader是我们的手段。

练习 3.3 注入多重依赖

在本节的例子中，通过使用 Reader monad 在一个领域服务中注入了一个仓储。很多情况下，会需要在领域服务中注入多个依赖。比如说，可能需要注入多个仓储、另外一个服务或者某些配置参数，作为领域服务的依赖。

为 AccountService 考虑一种合适的策略，用于处理多重依赖注入。一个方法是将所有依赖绑定为一个单独的环境类型，然后将其作为单独依赖进行传递。

本节的主要内容是展示如何使仓储可以通过上下文插入服务 API，而不是硬编码具体依赖。就像你看到的，使用 Reader 单子就是一个选择。我们还讨论了实现 Reader 的两个变体。在讨论的选项中，将仓储作为环境变量注入了计算。但还可以在更高水平的粒度（比如模块）注入依赖，那它们在函数级也将变得可用。这个手段会给整个模型的架构带来一个细微的变化，但今天很多人都在使用它。可以看看 MacWare（https:// github.com/adamw/macwire）以及它的设计哲学，它就是实现该策略的一个简单途径。[1] 本书主要使用 Reader monad 模式的变体在领域服务中注入仓储。

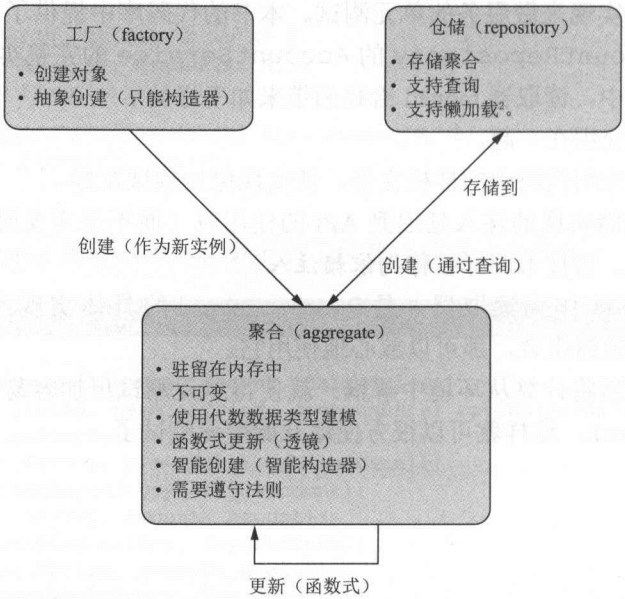


图 3.4 领域对象的三个生命周期模式之间的关系。聚合是这个链条中最重要也是最活跃的成员。它们存在于内存中，提供领域模型的大部分特性。仓储和工厂扮演支持者的角色。

1 参见Adam Warski所写的*DIinScala:Guide*，2016年（https://di-in-scala.github.io/#manual）。

2 用时加载而不是声明时加载。——译者注

3.3.7 高效地使用生命周期模式——结论

我们已经谈论了关于聚合、工厂的很多内容，以及应该如何在函数式世界中对待它们。下面将重点总结该设计模式的结论，以及关于替换实现技术的附加内容。图 3.4 中总结了 3 种模式的关系。

- 聚合是领域对象驻留内存的基本体现——包括实体和值对象。设计时需要非常小心，因为这些数据结构将成为领域 API 交互的基本点。
- 基于工厂的聚合的抽象创建，比如智能构造器，主要服务于两个目的：(a) 创建过程细节对用户抽象，(b) 如果一个类有多个子类型，可以将它们的创建都抽象在一个独立 API 中。
- 永远不要直接修改聚合。使用诸如透镜这样的数据结构来做函数式更新，同时还能提供可组合的 API。另一个可以用来处理聚合嵌套的数据结构是 *Zipper*。在这里我们不会讨论 Zipper 的实现，但可以从它的创建者 Gerard Huet 的原始论文 *The Zipper*（http://dl.acm.org/citation.cfm?id=969872）中获取一些灵感，或者看一下 Scalaz 或者 Shapeless 中的实现。
- 对于模式匹配来说，代数数据类型是很便捷的，如第 2 章中所述。模式匹配同样可以用来操作聚合。不过对于这种方式最好还是保守一点，毕竟有时候它会导致实现上一些无心的漏出。Scala 提供了一种提取器（extractor）的技术来解决该问题。[1]
- 时刻关注聚合所需要遵守的代数法则。在测试中将它们作为明确需要验证的属性。

在本节中，我们还了解了一些通用的函数式编程技术、抽象以及如何在领域建模实现中应用它们。在本书中，我们会生成越来越多的此类抽象以及其他类似的结构，所以必须理解这些抽象是如何工作的。如果现在还没有适应这个专题，就需要重新阅读本节直到真正掌握。图 3.5 总结了我们讨论的抽象，同时将它们与在领域建模中的运用方式做了映射。

1 更多细节参见Burak Emir等人的论文*Matching Objects with Patterns* (http://lampwww .epfl.ch/~emir/written/MatchingObjectsWithPatterns-TR.pdf)。

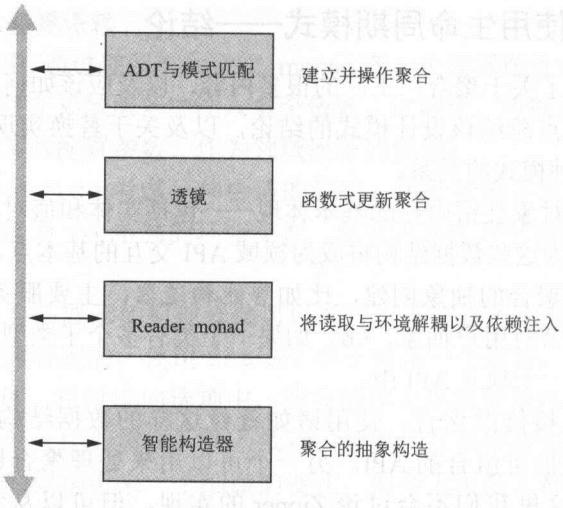


图 3.5 通用函数式编程抽象技术与领域建模中应用的映射。当我们在本书后续内容中讨论领域建模的其他元素时，可以看到此类抽象更多的使用场景。

3.4 总结

本章是我们第一次学到领域建模的基本模式，以及如何运用代数技术为领域模型设计 API。代数 API 设计是本书中递归主题的基本概念之一。以下是本章的几个主要结论。

- 代数思维：学到如何用模块所定义的代数的方式来思考 API 的设计。这与 OO 系统中设计 API 有很大的不同。焦点在于建模领域行为的操作以及如何将它们整合进模块中。
- 类型驱动的组合：首先确定每个 API 的代数，然后通过匹配类型和计算将它们捆在一起，组合成更大的行为。
- 关注隔离：需要将解释程序与代数解耦，我们也看到了一个完整的例子。这也是软件工程中关注隔离的一个案例。
- 聚合作为一致性单元：使用模块的代数法则来保证聚合的一致性边界。
- 领域对象模式的函数式实现：使用函数式编程的原则实现领域对象的生命周期模式，如工厂、仓储以及聚合。我们也学习了如何用代数数据类型设计聚合，用透镜更新聚合，设计仓储，以及用函数技术如 Reader 往领域服务中注入仓储。我们用工厂来创建对象，在 Scala 中用智能构造器来实现工厂。

4 领域模型的函数式模式

本章包括

- 理解函数式设计模式以及它们与 OO 设计模式的差异
- 代数作为模式
- 使用 monoid（幺半群），它是函数式编程中无所不在的设计模式
- 将模式用于作用编程（functor、applicative、monad）
- 使用了类型、代数与模式的两个案例，用更好的抽象建立领域模型

前面章节中介绍了代数 API 设计的技术，包括在系统中开始任何实现之前应该如何考虑 API 的组合。从现在起，当我们讨论 API 设计时，将首先聚焦在代数上。在本章中，将开始感受 API 代数中的模式。这些设计模式都是通用参数化的，而且不管建模的领域特性是什么，都可以在领域模型中重用它们。

对于那些已经使用过面向对象编程中设计模式的读者，本章将带大家进入函数式设计模式的世界——真正的可重用抽象，不必在每次用到它们时都重新实现一

遍。在本章中还将看到贯穿整个领域模型的计算结构的重用。我们将这些都称为设计模式，它们适合在某个特定上下文中解决一个特定的建模问题。我们将从代数和monoid的实现开始，这在函数式编程中是运用最广泛也是最有用的设计模式之一。接着我们将介绍被称为多作用的计算以及处理作用的一些模式，如applicative和monad。与这些模式一起，将看到它们的具体实现以及它们提供的组合器数目。这些组合器可以帮助我们通过领域模型的元素对这些模式进行组合。

图4.1展现了本章主题的主要路径。它将帮助我们在学习本章时有选择地学习自己关注的主题。

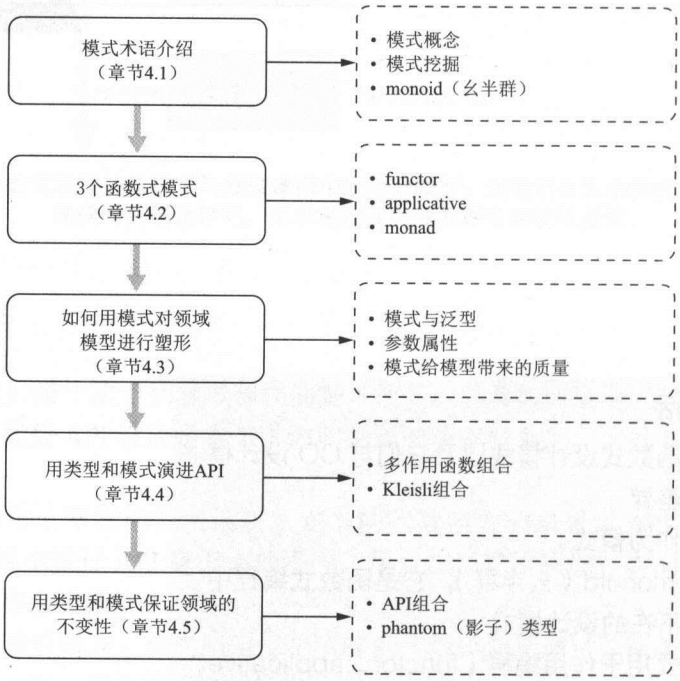


图4.1 本章学习过程。

学完本章，就能知道如何在领域中合理运用这些模式。

4.1 模式——代数、函数、类型的聚合

今天软件模式最通用的定义来自于James Coplien的*Software Patterns*（SIGS，1996年），它也是受到了架构大师Christopher Alexander的著作的启发：

> 一个模式就是一篇文学作品，它描述了一个设计问题以及在一个特定上下文中的该问题的通用解决方案。

Coplien 谈到了 3 件事：问题、解决方案和上下文。Alexander 在 *The Timeless Way of Building*（Oxford University Press, 1979 年）中描述模式的方式则更加简单明了：

> 每个模式都是由三块内容组成的一个规则，它反映的是某个上下文、问题与解决方案这三者之间的关系。

这个定义指出模式应该具备可重复性：解决方案往往在上下文中重复。从面向对象编程的经验来看，在 OO 中，术语设计模式已经基本上被制度化了。可重复性通常带有一点可重用性的意思。面向对象设计模式的现实设计概念（确切地说是一个标准词汇表）就是每次遇到一个问题和上下文时都必须分别实现。比如说，每次需要实现一个桥接（bridge）设计模式[1]来解耦抽象与实现时，几乎没什么具体实现是可以直接重用的。每个桥接都有一堆它自己的接口和实现，而我们需要将它们在特定的继承层级绑在一起。

本章中的函数式模式提供了比 OO 设计模式更好的可重用性。每个函数式模式都有两个完全独立的部分：

- 完全通用且可重用的代码，我们称其为代数[2]在所有使用模式的上下文中它都是不变的。
- 特定上下文的实现，在所有应用模式的实例中都各不相同。我们称之为代数的解释程序。

正如本章中所描述的那样，这是在建立可重用领域模型组件过程中闪闪发光的第一部分。

如果模式要在上下文中可重用，它必须具备足够的通用性，对上下文中的一般行为进行抽象。比如下面的抽象：

```
trait Monoid[T] {
  def zero: T
  def op(t1: T, t2: T): T
}
```

monoid 用类型 `T` 进行了参数化，这使我们在定义 monoid 时在类型上同样具备通用性。它提供了以下内容作为它的代数组成部分：

- 我们命名了 `zero` 的操作，对于下面的函数 `op` 来说，`zero` 将作为标识元素。这意味着用 `zero` 作为一个参数来调用 `op` 时，将总是返回其他参数。

1 参见Erich Gamma等人所著的*Design Patterns: Elements of Reusable Object-Oriented Software*（Addison-Wesley Profes- sional, 1994年）。

2 如果需要回顾一下代数相关的含义，可以参见3.1节关于代数API设计的内容。

- 我们还将一个二元的可结合的操作命名为 op。
- zero 的身份标识部分与 op 的组合性部分都完全遵守 monoid 的法则，如下所示：
  - 左标识：op(zero, t) == t
  - 右标识：op(t, zero) == t
  - 结合性：op(t1, op(t2, t3)) == op(op(t1, t2), t3)

monoid 的定义与它所满足的法则组成了代数。而且这也是可以在所有上下文中重用的代数。如果正在处理银行模型的一个特殊上下文，就可以定义一个将 Money 发布为 monoid 之前代数的实现。代数是不可变的，所以必须基于 Money 的上下文来解释 zero 和 op。

在 monoid 的定义中，没有任何关于类型的特定信息。代数使用的是一个通用的类型 T。这个概念在前面的章节中已经讨论过了。这就是*参数属性*的特性。在函数式编程中，好的模式在类型上都是参数化的，并通过它们定义的代数驱动可重用性。在章节 4.3 中将看到更多关于模式如何对领域模型进行塑形的内容。

4.1.1 领域模型中的挖掘模式

本节的模式都依赖于参数属性的优点。我们的目标是探索领域行为并识别出能够提炼为可重用模式的公共部分。在代码中，用来发现此类模式的方法如下：

- 收集用特定应用代码实现的特定案例。
- 在这些特定计算的结构与行为中寻找公共部分。
- 识别已存在的泛型抽象（模式，也就是代数[1]），用来对前面特定应用的计算进行建模。
- 用通用模式的实例替换特定应用代码。

领域模型中的抽象通常具有一些行为的共同点，当然也有不同。在个人银行领域中，假设有一个计算，过滤每天的高位值交易，还有一个计算，统计用户账户汇总的总余额。尽管这两个场景的计算细节是不同的，但依然还是存在相似之处，比如在这两个案例中都对交易或账户做了两个元素的二元操作。函数式模式有助于我们通过代数对公共部分进行统一，也允许我们对上下文实现的不同之处进行抽象。我们不要在抽象的术语上花费太多口舌，接下来看看领域中的具体案例，并试着挖掘出一些模式。

1 在前面的章节中已经说过，代数是模式的可重用部分。

4.1.2 用函数式模式使领域模型参数化

这里有一个个人银行领域的使用场景，它实现了后台管理功能。[1] 客户执行的所有借贷款形式的交易，都被记录在系统中，用于审核以及其他分析方面的需求。我们已经知道如何将一个客户的余额作为账户属性来管理。在这里将只考虑交易和余额，并尝试实现后台计算一个账户中所执行的不同交易的聚合。更特殊的是，在这部分模型中将实现以下行为：

- 在给定的交易列表中，将识别出在白天所发生的最高值的 debit 交易。这些值需要被高亮标识出来，以便于审核。
- 在给定的客户余额列表中，将计算所有 credit 余额的和。[2]

从领域逻辑的视角来看，所有的实现都很简单，因为实现的目的就是在函数式编程中识别出编程模式，而不用制定健壮的工业标准的模型。

识别公共部分

到目前为止，前面所有案例中的金额都定义为 `BigDecimal`[3]。但现实银行业务中，通常需要将货币与指定的金额关联起来。所以，是时候来充实模型的这个部分了。我们将开始新的 `Money` 模型，它同时具备金额与货币标签。不光如此，在你问我“有多少钱”时，我将检查钱包并说“我有 120 美元，25 欧元”。这也就意味着我们的 Money 抽象还要能够处理多种货币单位。清单 4.1 包含了将来用定义模块代数的 `Money` 和其他几个底层抽象。

清单 4.1 定义 `Transaction` 与 `Balance` 的底层抽象

```
sealed trait TransactionType
case object DR extends TransactionType
case object CR extends TransactionType

sealed trait Currency
case object USD extends Currency
case object JPY extends Currency
case object AUD extends Currency
case object INR extends Currency

case class Money(m: Map[Currency, BigDecimal]) {
  def toBaseCurrency: BigDecimal = //..
}
```

交易类型——可能是 debit（DR）或 credit（CR）

货币及其枚举

Money 抽象，用 Map 来编码多种货币的单位

1 这个案例实现的灵感来自于Runar在StackOverflow上对该问题的回答：http://stackoverflow.com/a/4765918。

2 credit余额也就是正余额。将忽略负余额（也被称为debit余额）。

3 用来对超过16位有效位的数字进行精确运算，通常用于商业计算。——译者注

```
case class Transaction(txid: String, accountNo: String, date: Date,
  amount: Money, txnType: TransactionType, status: Boolean)

case class Balance(b: Money)
```

客户在银行进行的交易

客户的余额

定义的行为属于一个特殊模块（比如说 Analytics）。清单 4.2 提供了一个包含解释程序的模块代数的案例。

注意：实现的一些细节没有体现在清单 4.2 中，但这并不妨碍我们理解它的本质。完整的代码可以在本书的代码库中找到。

清单 4.2 模块 Analytics 的代数与实现

```
trait Analytics[Transaction, Balance, Money] {
  def maxDebitOnDay(txns: List[Transaction]): Money
  def sumBalances(bs: List[Balance]): Money
}

object Analytics extends Analytics[Transaction, Balance, Money] {
  def maxDebitOnDay(txns: List[Transaction]): Money = {
    txns.filter(_.txnType == DR).foldLeft(zeroMoney) { (a, txn) =>
      if (gt(txn.amount, a)) valueOf(txn) else a
    }
  }
  def sumBalances(balances: List[Balance]): Money = {
    balances.foldLeft(zeroMoney) { (a, b) =>
      a + creditBalance(b)
    }
  }
  private def valueOf(txn: Transaction): Money = //..
  private def creditBalance(b: Balance): Money = //..
}
```

模块的代数

省略了根据 Money 定义的分类，详见代码库

省略了 Money 的加法，详见代码库

当它是 credit 余额时返回余额值，否则返回 0

用 valueOf 获取交易的货币值

在 maxDebitOnDay 和 sumBalances 行为的实现中，有没有看到一些相似的地方，可以重构出更通用的模式？我们在这里列出一些：

- 这两个实现都折叠了集合来计算核心领域逻辑。
- 折叠将 Money 的一个单元对象作为累计的种子，然后执行基于 Money 的二元操作进行循环累加。在 maxDebitOnDay 中，这个操作是做比较；在 sumBalances 中，该操作是做加法。它们确实不同，但又都是可结合的、二元的。

大家已经看到我们的重头戏在哪了——monoid（幺半群）。这是这个练习中最重要的部分：观察模式并识别出合适的代数。不可能每次都正好合适，有时需要调整一下实现来适应它。不过这绝对是值得的。因为不需要去实现现有代数错误百出的变体，只需要直接重用它就行了。这些模式已经经过专家们的千锤百炼，也在不同的产品实现中经受了多次现场测试。下一步是使用 monoid 的代数来统一这两个看

起来不同的操作。

操作的抽象

接下来定义 Money 的 Monoid 实例。因为已经用 Map 的方式定义了 Money，所以先要定义 Monoid[Map[K, V]]，然后用它定义 Monoid[Money]。事实上，我们需要定义两个 Monoid[Money] 实例，因为有 maxDebitOnDay 和 sumBalances 两个不同的操作需求。前者需要实例来做 Money 的比较操作，而后者需要另一个实例来做加法操作。这里只演示加法的例子，基于比较的实例实现起来有点冗长，具体代码可查阅代码库。

```
implicit def MoneyAdditionMonoid = new Monoid[Money] {
  val m = implicitly[Monoid[Map[Currency, BigDecimal]]]
  def zero = zeroMoney
  def op(m1: Money, m2: Money) = Money(m.op(m1.m, m2.m))
}
```

如果 V 是 monoid，那 Map[K, V] 也是一个 monoid。这里的 BigDecimal 就是一个 monoid。

清单 4.3 展示了 Analytics 模块的实现，其中的 Money 使用了 monoid。[1] 这是使模型更加通用的第一步。折叠内的操作现在是一个 monoid 操作，而不是针对特定领域类型的硬编码的操作。

清单 4.3 通过 monoid 抽象操作

```
trait Analytics[Transaction, Balance, Money] {
  def maxDebitOnDay(txns: List[Transaction])
    (implicit m: Monoid[Money]): Money
  def sumBalances(bs: List[Balance])(implicit m: Monoid[Money]): Money
}

object Analytics extends Analytics[Transaction, Balance, Money] {

  def maxDebitOnDay(txns: List[Transaction])
    (implicit m: Monoid[Money]): Money = {
    txns.filter(_.txnType == DR).foldLeft(m.zero) { (a, txn) =>
      m.op(a, valueOf(txn))
    }
  }

  def sumBalances(balances: List[Balance])(implicit m: Monoid[Money]): Money =
    balances.foldLeft(m.zero) { (a, bal) => m.op(a, creditBalance(bal)) }
```

对 Money 来说，两个函数都需要一个隐式 Monoid。这将被用于将 Money 的特定操作转变为 Monoid 操作的实现中。这也使得这些操作可以被任意 Monoid 所重用，而不仅限于 Money。

1 清单4.3（包括本章其他很多例子）包含了Scala中一些高级的句法风格。如果需要温习这部分内容，请参考Dean Wampler和Alex Payne所写的*Programming Scala*（O'Reilly Media, 2014年）。要了解函数式编程风格，参见*Functional Programming in Scala*。

```
  private def valueOf(txn: Transaction): Money = //..
  private def creditBalance(b: Balance): Money = //..
}
```

Scala 贴士

Scala 中需要明确地传递一个隐式对象给每一个需要它的函数，请留意清单 4.3 中是如何为函数实现相同功能的。这也被称为明确字典传递技术：将隐式字典和函数定义一起传递。这种做法的好处是可以定义 Monoid[Money] 的多重实例（事实上马上就会这样做），而编译器只会传递合适的实例给函数，但需要确保通过隐式声明的规则和范围，编译器可以选取到正确的实例。如果在范围中有多重隐式，必须明确地传递我们所希望的那个。[1]

在我们的例子中，需要有两个实例，一个实例用于 maxDebitOnDay 比较 Money 的两个实例，另一个用于 sumBalances 做 Money 的累加。当调用操作时，必须传递正确的实例。

上下文的抽象

在之前的实现中，我们发现 maxDebitOnDay 和 sumBalances 折叠操作内的行为是很类似的。在这两个场景中，已经抽象了传递的 monoid 操作。代码因为这个抽象而变得更加通用，同时也降低了对特定领域元素知识的理解要求。

如果对这两个函数观察得足够仔细，可以发现另外一个相似点。在这两个案例中，在生成 monoid 的函数映射之后折叠了一个集合。比如 maxDebitOnDay 中，用 valueOf 做映射，表示 Transaction => Money。而 sumBalances 中用了 creditBalance，表示 Balance => Money。而 Money 就是一个 monoid。[2] 如果集合包含的元素本身就是 monoid，就不需要做任何映射（确切地说可以和一个标识函数做映射）。总的来说，我们在这两个函数中所做的是：给定一个可以折叠的集合 F[A]，对 F[A] 做折叠，这里要么 A 是一个 monoid，要么可以被映射到一个 monoid。该集合所需要的唯一属性就是它的可折叠能力。因此，可以将集合定义为 Foldable[A]，这样它就可以有更好的通用性（当然威力也有所减弱）；在这里并不需要 List[A] 那么丰富的功能。以下是 Foldable 类型构造器的代数定义：

1 更多关于Scala中明确目录传递技术的细节，参见Bruno Oliveira等人的论文*Type Classes as Objects and Implicits*（http://ropas.snu.ac.kr/~bruno/papers/TypeClasses.pdf）。

2 说A是一个monoid，即A是一个类型，它有一个定义的monoid实例。

```
trait Foldable[F[_]] {
  def foldl[A, B](as: F[A], z: B, f: (B, A) => B): B
  def foldMap[A, B](as: F[A])(f: A => B)(implicit m: Monoid[B]): B =
    foldl(as, m.zero, (b: B, a: A) => m.op(b, f(a)))
}
```

函数 foldMap 实现了之前所说的：折叠集合 F[A]，f:A=>B 将根据 A 生成一个 monoid B。同时，如果 A 是一个 monoid，那么 f 可以是一个标识函数。所以，用一个 Foldable[A]，一个 monoid 的类型 B，一个 A 和 B 之间的映射函数，就可以将 foldMap 打包进一个组合器来抽象对 maxDebitOnDay 和 sumBalances 的需求（包括其他类型的领域行为），同时也不用牺牲参数属性的法宝。这也是通过设计模式使模型更加通用的第二步：将对上下文——抽象的类型构造器——进行抽象。

```
def mapReduce[F[_], A, B](as: F[A])(f: A => B)
    (implicit fd: Foldable[F], m: Monoid[B]) = fd.foldMap(as)(f)
```

现在每个模块函数都变得像个单行小程序：

```
object Analytics extends Analytics[Transaction, Balance, Money] {
  def maxDebitOnDay(txns: List[Transaction])
    (implicit m: Monoid[Money]): Money =
    mapReduce(txns.filter(_.txnType == DR))(valueOf)

  def sumBalances(bs: List[Balance])(implicit m: Monoid[Money]): Money =
    mapReduce(bs)(creditBalance)
}
```

这个练习的完整可运行代码可以在本书的代码库中找到。

最后，从使用 Monoid 或 Foldable 这种设计模式中，我们学到了些什么呢？

- *更通用*：领域行为现在用完全通用的 mapReduce 函数来实现，这也提升了模型的抽象水平。同时因为 mapReduce 彻底通用，它可以作为一个解决方案被重用于其他类似问题。通过模式挖掘的练习，对两个函数式模式（Monoid 和 Foldable）的代数进行统一，找到泛型抽象 mapReduce。
- *更抽象*：已经可以用 Monoid 设计模式对操作进行抽象，用 Foldable 模式对类型构造器进行抽象。这使模型更加模块化，同时更容易管理和验证。
- *平行化*：最后，因为 monoid 式的操作都是可结合的，那么操作就可以是平行的（所有可组合操作都是如此）、增量计算的，而且是可以缓存的。[1]

1 关于平行化的更多细节，参见Murray Cole的文章*Parallel Programming with List Homomorphisms*，发表于Parallel Processing Letters 5:191-203, 1995年。

4.2 强类型函数式编程中计算的基本模式

如果使用强类型函数式编程语言，领域建模的最重要的一个方面就是采用类型系统的计算结构的组织，这样在模型中就可以达到最大的组合性。记住，当类型匹配时抽象才可以组合；否则就会被遗弃在孤岛上，既不能重用也不能组合。在这方面已经有相当多的研究，特别是在 Haskell 社区，人们已经发现，通过强类型函数组合建立抽象，可以很接近地映射到范畴理论中的组合态射（morphism）。范畴理论中的很多模式，比如 functor 和 monad，在强类型函数式编程的世界中也找到了类似的存在。本书并没有包含范畴理论的内容，但会触及到这些话题，我们只是想强调本节这些模式的理论根源来自于范畴和态射的数学基础理论。

本节中的模式奠定了函数式编程多作用计算的基础。它们提供抽象在领域模型内处理各种作用。[1] 在现实生活中设计一个领域模型时，会发现很少有操作在本质上是纯粹的。在大部分的行为里，需要实现各种作用，比如处理可选值、异常跟踪以及 IO 等。使用本节的模式将帮助我们在多作用计算之上设计一个纯粹的引用透明的用户接口。在第 3 章中已经有类似的例子：在 Scala 中用 `Try` 的抽象在设计 `AccountService` 模块的接口时，如何处理异常。不过 `Try` 是一个特殊的案例，在本章中，将学到如何用代数的方式归纳作用，也将看到通用的模式，如 functor、monad 以及 applicateive，用它们开发一个通用的计算结构来抽象及组合作用。在本节的最后，将谈到如何在领域模型中描述作用，如何将它们组装到一起形成更大的作用。

4.2.1 函子——建立模式

大家是否见过 Scala 标准库中一些类的 `map` 的特征，如 `List`、`Option` 或 `Try`？如果没有，下面的代码片段将它们放到一起进行呈现：

```
// map for List[A]
def map[B](f: (A) => B): List[B]

// map for Option[A]
def map[B](f: (A) => B): Option[B]

// map for Try[A]
def map[B](f: (A) => B): Try[B]
```

一个模式将反复使用它们。如果忽略应用的上下文，这些函数都有相同的特征。就函数式编程而言，将该上下文称为一个作用。`List[A]` 提供了类型 A 元素的重复作用。`Option[A]` 建模了不确定的作用，其中类型 A 的元素既可以存在也可以

1 在第2章中已经讲过如何应对多作用计算。

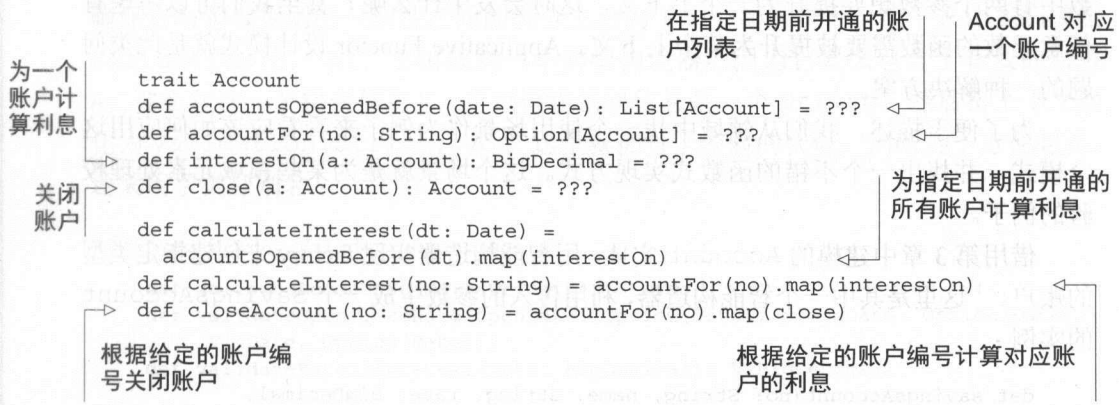
不存在。Try[A] 提供了异常具体化的能力。所以，如果可以从这些特征中抽象出作用，我们所拥有的就是一个函数 map，它将函数 f 提升为一个作用 F[_]。每一个作用都是用类型构造器建模：-F[_] 不单单是个 F，可以将这个计算结构提取为一个独立的 trait 并称之为 Functor：

```
trait Functor[F[_]] {
  def map[A, B](a: F[A])(f: A => B): F[B]
}
```

因为 List、Option 等都共享这个计算行为，所以可以分别为它们实现一个 functor 代数的解释程序，使它们都变化成 functor：

```
def ListFunctor: Functor[List] = new Functor[List] {
  def map[A, B](a: List[A])(f: A => B): List[B] = a map f
}
def OptionFunctor: Functor[Option] = new Functor[Option] {
  def map[A, B](a: Option[A])(f: A => B): Option[B] = a map f
}
```

可以说一个 functor 基于一个数据结构和二元函数抽象了 map 的能力。你现在可能会问，这样做给领域建模的上下文带来了什么好处？答案很简单：它归纳了模型，并通过组件使模型行为的某些部分变得可以重用。让我们来看一个例子，假设实现的一个模块中有以下函数：




```
trait Account
def accountsOpenedBefore(date: Date): List[Account] = ???
def accountFor(no: String): Option[Account] = ???
def interestOn(a: Account): BigDecimal = ???
def close(a: Account): Account = ???

def calculateInterest(dt: Date) =
  accountsOpenedBefore(dt).map(interestOn)
def calculateInterest(no: String) = accountFor(no).map(interestOn)
def closeAccount(no: String) = accountFor(no).map(close)
```

在函数 calculateInterest 和 closeAccount 中，通过所提供的计算利息或关闭账户的函数，对具体的指定应用的数据结构做了 map。使用 functor，可以将这种映射行为提炼成一个通用函数，它根据提供 functor 实现的类型进行参数化。参数化之后，这个函数使代码降低了领域类型的相关性，同时也提供了一个可重用的函数，它可以用于任何我们想要映射一个 functor 的地方。这也正是在之前图 4.1 中所描绘的场景。仅仅用 Functor 替换 Monoid 的上下文，就可以拥有之前所描述

的使用场景：

```
def fmap[F[_], A, B](fa: F[A])(f: A => B)(implicit ft: Functor[F]) =
  ft.map(fa)(f)
```

现在可以用 fmap 来实现特定的领域行为：

```
def calculateInterest(dt: Date) =
  fmap(accountsOpenedBefore(dt))(interestOn)
def calculateInterest(no: String) =
  fmap(accountFor(no))(interestOn)
def closeAccount(no: String) = fmap(accountFor(no))(close)
```

Functor 使我们具备了基于纯函数映射的能力，这也是它们所能做的全部。不用奇怪，大家很快就会发现只有少量的使用场景需要对纯函数进行映射。Functor 的主要作用是提供了一个途径，通向下一个多作用计算的模式——加强版的 functor 和 monad。

4.2.2 加强版函子模式

Functor 使我们具备将一个参数的纯函数提升为特定上下文的能力。可以观察 trait Functor 中 map 的定义。纯函数 f:A=>B 被提升为 F 的上下文，从 F[A] 转变到 F[B]。这里的上下文就是*类型构造器* F，也被称为计算的*作用*。但如果一个函数中有两个参数想要提升为一个上下文，这时会发生什么呢？甚至我们可以考虑有任意元数的函数需要被提升为某个上下文。Applicative Functor 设计模式就是此类问题的一种解决方案。

为了便于描述，我们从领域中找一个使用场景作为例子来看看应该如何应用这个模式，并找出一个不错的函数式实现方式。这个场景就是为某些领域元素处理校验的例子。

借用第 3 章中建模的 Account 实体，用智能构造器实现 factory 来创建指定类型的账户。[1] 这里是其中一个智能构造器，利用传入的参数生成一个 SavingsAccount 的实例：

```
def savingsAccount(no: String, name: String, rate: BigDecimal,
  openDate: Option[Date], closeDate: Option[Date],
  balance: Balance): Try[Account] = {

  closeDateCheck(openDate, closeDate).map { d =>
    if (rate <= BigDecimal(0))
      throw new Exception(s"Interest rate $rate must be > 0")
    else
```

1 参见第3章中讨论的智能构造器。

```
        SavingsAccount(no, name, rate, Some(d._1), d._2, balance)
    }
  }
```

除了构造部分，智能构造器还在账户的开户和关闭日期之间执行了一个一致性校验。这里还可能有其他校验——比如，检验账户编号是否是特定格式，或者所提供的利率是否合法。通常来说，校验逻辑都是比较复杂的，在返回一个合法的聚合给用户之前，通常需要调用多个校验。如果是命令式风格的编程，可以使用嵌套构造，直到最后完成校验。但在函数式范式中，我们更喜欢面向表达式的编程。[1] 我们应该将所有校验组合在一起成为一个表达式，当校验都被满足时，它将返回一个有效的构造聚合。

不管前面的定义和实现，让我们试着建立一个认知，如何将混合校验演变为一个抽象，以及如何将其归纳为一个设计模式。这种探索方式不仅仅让我们知道建立模型应该是什么样的，还告诉我们如何根据零碎的需求增量式地建立一个模型。根据国际惯例，我们将用一个代数模型来建立合适的 API。需求包含两个方面：

- 每个校验函数需要返回一个上下文，它要么包含有效的对象，要么包含一个错误说明为什么会校验失败。
- 这些上下文要么被用来构建 SavingsAccount 的实例，要么用来报告错误给客户。

通过引入抽象了校验结果的上下文，我们已经在 SavingsAccount 的构造和确保它与领域规则的一致性之间的拆分上达到了一个新的水平。接下来让我们调用 Validation[E, A]，它将成为一个类型构造器，因为它要么包含了有效的对象 A，要么包含一个错误信息 E。可以假设错误信息是一个 String，然后将上下文简化为 Validation[String, A]。这里就拥有了校验的代数，同时为了方便起见，还采用了类型别名：

```
type V[A] = Validation[String, A]

def validateAccountNo(no: String): V[String]
def validateOpenCloseDate(openDate: Option[Date], closeDate: Option[Date]):
    V[(Date, Option[Date])]
def validateRateOfInterest(rate: BigDecimal): V[BigDecimal]
```

我们现在对 Validation 的实现依然一无所知——只是一个类型的占位符，它如何应用于校验函数的代数呢？在调用全部 3 个校验函数之后，将得到 3 个 V[_] 的实例。如果结果都是校验成功，就提取有效的参数并传给函数 f，它将构造最终的合法对象。如果遇到失败，就将错误报告给客户。下面就是该工作流的具体内容——

1 第1章和第2章都谈到了面向表达式编程。

我们称其为 apply3。[1]

```
def apply3[V[_], A, B, C, D](va: V[A], vb: V[B], vc: V[C])
  (f: (A, B, C) => D)
  : V[D]
```

输入上下文

处理函数

在相同上下文中的有效输出对象

通过另外一个方式来看这个函数，这也会使我们在抽象上有不同的理解：

```
def lift3[V[_], A, B, C, D](f: (A, B, C) => D)
  : (V[A], V[B], V[C]) => V[D]
  = apply3(_, _, _)(f)
```

在这里，将纯函数 f 变为上下文 V。试想一下，在什么场合，我们会更喜欢这种变体，而不是使用 apply3？练习 4.1 会讨论这方面更多的细节。

练习 4.1 代数化思考

在本练习中，我们将了解代数化设计，也将看到代数的某个形态有时比同样代数的另一种表达形式更加有用。本节中，不管 apply3 还是 lift3 的定义都实现了同样的目的。它们结合 3 个参数的纯函数和 3 个校验上下文，帮我们对每个参数做校验。但两者的用法是不同的，同时针对如何运用函数到上下文也会给我们带来不同的思路。

仔细观察 apply3 和 lift3 的特征。apply3 将 3 个校验上下文和 3 个参数的纯函数 f 都作为输入。它将函数应用到 3 个上下文 V[A]、V[B] 和 V[C] 中，并且将上下文 V[D] 返回给我们。该结果也是最终赋值的上下文。

而另一方面，lift3 没有包含函数 f 的应用作为它实现的组成部分，而是返回一个抽象，当它赋值时给我们相应的校验上下文。它在上下文参数上是咖喱化的。

请分析以下这两种途径的优势和劣势，以及在什么情况下偏向使用哪一种。

提示：阅读 3.2.1 节，考虑抽象带来早期赋值的好处。看起来 lift3 比 apply3 组合得更好。

这时我们还没有 apply3 的实现，但假设已经有了，让我们看看如何从代数演进出校验代码，并插入到智能构造器中，创建一个 SavingsAccount：

```
def savingsAccount(no: String, name: String, rate: BigDecimal,
  openDate: Option[Date], closeDate: Option[Date],
  balance: Balance): V[Account] = {
```

1 apply意味着我们在应用上下文到函数。而3表明了上下文的数目。

```
    apply3(
      validateAccountNo(no),
      validateOpenCloseDate(openDate, closeDate),
      validateRate(rate)
    ) { (n, d, r) =>
        SavingsAccount(n, name, r, d._1, d._2, balance)
    }
  }
```

校验的 3 个上下文

函数从上下文中获取信息并构造一个有效的 SavingsAccount

validateOpenCloseDate 返回一个二元组，用 d._1 和 d._2 访问两个成员

apply3 能很好地用于这个使用场景。但需要将整个工作流归纳进一个抽象，这样才能形成一个广泛的应用。apply3 或 lift3 终究不是专门校验字段的 API。apply3 或 lift3 将存在于何处？现在它们被安置在全局命名空间，我们用上下文类型构造器 V[_] 对它们进行参数化。让我们把它们转移到一个模块里，同时挖掘出其中的模式。

我们刚才已经看过了函数式编程的 *Applicative Functor*。在面对函数式编程中的 *applicative* 作用（不久就会看到另外一种作用类型，单子作用）时，它是非常有用的模式。就像其名字所说的那样（加强版函子），applicative functor 是在 functor 之上建立了额外的能力，而属于 *applicative* 指的是作用被应用的方式。如果观察 apply3，可以发现不管每个作用可能产生的结果是什么，都会通过所有参数被序列化。比如在之前的应用中，所有的 3 个校验函数都会被执行，不管每一个是成功还是失败。在 Paul Chiusano 和 Runar Bjarnason 所著的 *Functional Programming in Scala* 一书中包含了该模式的所有细节，在这里就不再重复了。要了解 applicative functor 实现的更多细节，请参考 Scalaz（https:// github.com/scalaz/scalaz）。对于还不了解 Scalaz 的读者，在后面的贴士中会讲到该库的一些特性。

清单 4.4 Applicative Functor 特征（简化版）

```
trait Applicative[F[_]] extends Functor[F] {
  def ap[A,B](fa: => F[A])(f: => F[A => B]): F[B]

  def apply2[A,B,C](fa: F[A], fb: F[B])(f: (A, B) => C): F[C] =
    ap(fb)(map(fa)(f.curried))

  def lift2[A,B,C](f: (A, B) => C): (F[A], F[B]) => F[C] =
    apply2(_, _)(f)

  def unit[A](a: => A): F[A]
}
```

需要提供实现类的原始操作。马上就会看到实现的例子。

在拥有 Applicative trait 之后，准备一个 Applicative 实例用于 Vali-

dation。[1] 同时，还得到了 savingsAccount 的实现，它包含了用 applicative 作用实现的校验逻辑。

```
val av: Applicative[V] = …
def savingsAccount(no: String, name: String, rate: BigDecimal,
  openDate: Option[Date], closeDate: Option[Date],
  balance: Balance): V[Account] = {

  // ..

  av.apply3(
    validateAccountNo(no),
    validateOpenCloseDate(openDate, closeDate),
    validateRate(rate)
  ) { (n, d, r) =>
      SavingsAccount(n, name, r, d._1, d._2, balance)
  }
}
```

从 V 的 Applicative 实例中调用 apply3。

Scalaz 库提供了一个完整的可用于生产的 Applicative 实现。本书的代码库，在用 Scalaz 提供的 Validation 抽象实现的智能构造器中包含了 Account 校验的完整实现。

Scalaz——简要介绍

Scalaz 是一个库，它用 Scala 编程语言实现了纯函数式的抽象。它与 Haskell 提供的抽象在精神上非常相似，即利用参数化多态性的力量来组合并扩展它们。以下就是 Scalaz 所提供的特性清单：

- Scalaz 的基本抽象是类型类，它允许特定的多态性用于扩展。Scalaz 提供了几乎所有在函数式编程中可能会需要的抽象的类型类。想要全面了解类型类的概念和分级，请参考 Typeclassopedia。
- 除了一系列的类型类，Scalaz 还提供了纯函数式的数据结构，比如 Finger Search Tree、Difference List、NonEmptyList、Lenses 和 Zippers。
- 和 Haskell 一样的控制作用的方式。
- Monad 转换器，比如 ReaderT、WriterT 和 StateT，可以用它们来组合 monad。
- 可组合的强类型 actor。

1 用于Validation的Applicative需要类型的局部应用。事实上，它会有一个Applicative[({type a[x] = Validation[E,x]})#a]的类型。作为用于V的Applicative来说这样更容易编写，就如同alidation [String, A]这样。*Functional Programming in Scala*中提供了Scala局部类型应用的更多细节。

- 它还对 Scala 标准库中的很多类做了增强，比如 List 和 Option，使它们具备更多的函数特性。它用改进过的变体替换了标准库中的一些抽象（比如，它提供了一个 Disjunction 类型（\/），这是 Either 的一种实现，但它比标准库提供的那个有用得多）。

关于 Scalaz 的更多细节，参见 https://github.com/scalaz/scalaz。

为什么将泛型模块用于 Applicative

大家一定已经注意到，我们为 applicative 定义了一个泛型模块。为了获取一个任意数据类型的 Applicative，需要为该数据类型定义一个具体实现，它将实现泛型抽象提供的抽象函数。比如，要为 List 生成一个 Applicative，需要像下面这样定义一些东西：

```
def ListApply: Applicative[List] = new Applicative[List] {
  def map[A, B](a: List[A])(f: A => B): List[B] = a map f
  def unit[A](a: => A): List[A] = List(a)
  def ap[A, B](fs: List[A => B])(as: List[A]): List[B] = for {
    a <- as
    f <- fs
  } yield f(a)
}
```

ListApply 实现了一个针对 List 的 Applicative 实例，这就意味着能够对 List 的元素实现作用的序列。这使得 Applicative 成为一个开放的抽象，可以事后再扩展任意抽象，使其成为一个 Applicative。这也是与通过子类型进行扩展的典型区别所在，就子类型的方式来说，当定义类型时，需要事先定义一个类作为子类型。

泛型 applicative（或 functor）定义带来的另一个好处就是在它们之上定义泛型组合器的能力。这里是 Scalaz 里的例子：[1]

```
def traverse[F[_]: Functor, G[_]:Applicative, A, B](fa: F[A])
  (f: A => G[B])(implicit 1: Foldable[F]): G[F[B]] = …
```

观察这个函数的类型，就会发现它提供了一个方式，可以对一个 Foldable[2] 序列做多作用的遍历。这也是该函数在我们的领域模型中的典型应用。比如说有一系列账户的列表，其中有些可能已经被关闭了。我们需要写个函数，如果没有账户被

1 Scalaz把traverse定义成独立的类来做遍历；在这里我做了一些简化。

2 在章节4.1.2中我们讨论过Foldable。

关闭，那就返回这些账户的余额列表。但只要有账户被关闭了，这个函数就不做任何动作，直接返回一个类型说明此事即可。

```
import scalaz._
import Scalaz._
val accounts: List[Account] = //..
Applicative[Option].traverse(accounts) { a =>
  if (a.dateOfClose isDefined) None else a.balance.some
}
```

这里将 Option 用为 applicative，并将计算结果存入 Applicative[Option] 内。如果没有账户被关闭，所有作用都将被序列化，而且我们会在 Option 里取到最终的列表，否则就会得到一个 None。作为一个领域建模人员，将从这种泛型组合器中获取巨大的力量。注意，traverse 就是基于类型参数和约束彻底地参数化，这些参数和约束被定义为代数的组成部分。如刚才所见，通过合适的类型特化，就可以实现特定领域的遍历功能。

Applicative Functor 模式给领域模型带来的好处是什么

我们现在已经看到模式是如何通过领域建模的特定使用场景来进行演进的，接下来再从泛型模式的视角来讨论它所带来的帮助。大家一定已经意识到，这个模式背后的核心思想是作用的序列化。我们有一串上下文（作用），在我们的例子中，这些都是校验函数，它们会返回一个 Validation 的实例。这里的主要思想是，不管每个生成的结果是什么，所有这些上下文都将被赋值。某些校验可能会失败，但并不会阻止其他校验的执行。当映射一个 applicative functor 时所有上下文都会被独立赋值。只有当它们都成功的时候，函数才会创建新的上下文。

因此，如果需要执行那些在序列中彼此独立的上下文时，可以用 Applicative Functor 模式。其计算的形态就是所有上下文都得到执行[1]，任何执行都不依赖其他上下文是否执行成功。这是 applicative functor 和 monad 的区别。图 4.2 将一个 applicative 描述为作用的序列。

1 计算的形态被保留在applicative作用中，不管它们的输出是什么，所有的输入都将得到处理。而在单子作用中，一个计算可以决定其他计算是否继续进行。在一个单子应用中有可能不是所有输入都会得到计算。

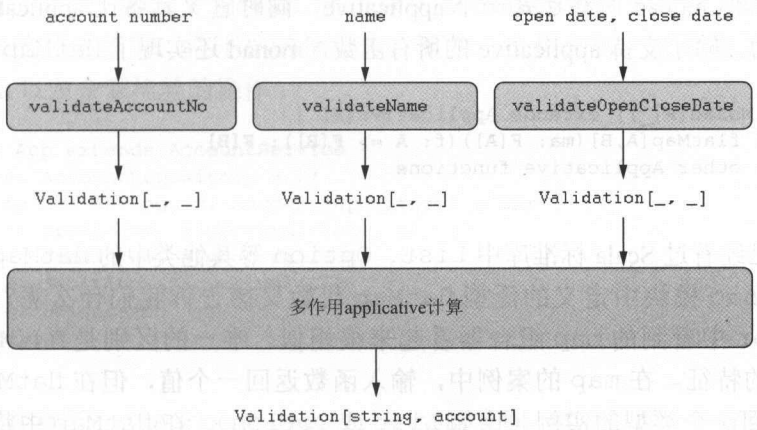


图 4.2 Applicative 计算保留了计算的形态。在通过 applicative 计算之前，不管是否有发生失败，所有校验都会被计算。

applicative functor 对于 validation 这种使用场景来说是非常适合的。另一个例子是执行一串互相独立的批处理然后计算结果。事实上，所有的处理都是独立执行的，所以也可以考虑通过并行的方式来进行优化（也许可以用基于 `Future` 的计算）。这也是该模式带来的另一个好处：因为我们知道所有的上下文都会被执行，所以可以事先为它们制定好赋值策略。

4.2.3 单子作用——applicative 模式的变体

大家现在应该已经很熟悉 monad 了。第 3 章中，在函数式组合器的上下文、用 `flatMap` 序列化作用以及向领域服务中仓储的依赖注入都涵盖了 monad。但每个例子都引用了特定的类型构造器（如 `List` 或 `Option`）来实现 `flatMap`。在本节中，将学到如何从这些特化中建立一个通用模式，使用 applicative 来建立一个 `Monad[T]`，并结合所需的代数来实现期望的作用。作用的语义会与 applicative 中的不同，我们将看到如何使用不同的建模上下文中的语义。

本节将从以下 4 个方面来讲述 monad：

- 单子作用与 applicative 作用的不同之处。
- 结合 Scalaz 中的实现学习泛型 monad 模块。
- 一个 monad 特定类型的案例——`State` monad（用非特定类型构造器实现一个单子作用），以及如何在领域模型中高效地使用它。
- 如何使用泛型 monad 组合器实现领域模型的特定功能。

泛型 Monad 模块

Scala 里的泛型 monad 模块看起来是什么样的？ monad 实现了与 applicative 不

同的语义，但 monad 同样也是一个 applicative，同时它又具备比 applicative 更强大的抽象能力。除了支持 applicative 的所有函数，monad 还实现了 flatMap：

```
trait Monad[F[_]] extends Applicative[F] {
   def flatMap[A,B](ma: F[A])(f: A => F[B]): F[B]
  //.. other Applicative functions
}
```

我们已经看过 Scala 标准库中 List、Option 等其他类中的 flatMap 方法。但在之前 Monad 模块中定义的泛型 flatMap 代数又想告诉我们什么呢？它与我们在 Functor 中看到的 map 组合器看起来很相似。唯一的区别是 flatMap 所获取的函数 f 的特征。在 map 的案例中，输入函数返回一个值，但在 flatMap 中它为 monad 返回一个类型的实例（比如 List 或 Option）。在 flatMap 中将函数 f 应用为类型 F[A] 的输入 ma 时，将以 F[F[B]] 作为结束。flatMap 将这两者连接起来，将它们扁平化成一个单独的作用 F[B]。从语义上来说，flatMap 等同于扁平化的 map。[1]

另一个关于 flatMap 很重要的点就是将多个 flatMap 组装到一起时，这个计算链是如何进行的。它们序列地组合在一起，其中一个 flatMap 发生中断，则整个链条也随之中断。这与 applicative 的工作方式是不同的。下面也会看到如何利用这个特性来实现领域建模中需要快速失败的场景。

单子作用与 Applicative 作用有怎样的不同？

在前面章节中已经介绍过 applicative 作用是如何保留计算的形态的。因为我们预先知道计算的步骤，所以可以进行一些优化，比如为了获取更好的吞吐量将其中一些计算并行进行。在单子作用的场景中，情况就会有些不同。

考虑如下 AccountService 模块的代数，它包含返回 Try[_] 的方法，以应对操作可能失败的情况：

```
trait AccountService {
  def open(no: String, name: String, openDate: Option[Date],
    r: AccountRepository): Try[Account]
  def close(no: String, closeDate: Option[Date],
    r: AccountRepository): Try[Account]
  def debit(no: String, amount: Amount,
    r: AccountRepository): Try[Account]
  def credit(no: String, amount: Amount,
    r: AccountRepository): Try[Account]
  def balance(no: String, r: AccountRepository): Try[Balance]
}
```

1 关于如何实现monad的完整解释，参见*Functional Programming in Scala*。

我们知道，Try 是一个 monad，也可以从 AccountService 中串联多个操作来建模更大的领域行为。在下面的例子中，就在一个用户账户上串联了一系列的 credit、debit 以及余额检验的操作：

```
object App extends AccountService {
  val r: AccountRepository = //..
  def op(no: String, r: AccountRepository) = for {
    _ <- credit(no, BigDecimal(100), r)
    _ <- credit(no, BigDecimal(300), r)
    _ <- debit(no, BigDecimal(160), r)
    b <- balance(no, r)
  } yield b
}
```

假设已经有了一个 AccountRepository 的实例，然后根据一个存在的账户编号（就是说账户在存储中是存在的）调用了 op。op("accountNumberExists", r) 将导致 op 实现中对所有 debit、credit 以及 balance 的调用都会被执行一遍。我们也知道，flatMap 和 map 通过完成计算的整个链条对作用进行排序。

现在来考虑其他的情况：如果给 op 传递一个存储中不存在的账户编号会怎样？ op("accountNumberNotExists",r) 会在第一步就立刻产生一个失败，对 credit(no,BigDecimal(100),r).flatMap { _ => .. 的调用返回一个 Try[Account]，结果是 Failure，于是 flatMap 的整个链条在这里就断开了。后续对 credit、debit 以及 balance 的调用永远都不会被执行。而图 4.2 中 applicative 作用是不管其他人的结果是什么，所有的校验函数都会被执行一遍。所以单子作用在计算中导致的结果就是它是受条件约束的，每一步的执行都依赖于链条中的上一步成功与否。[1] 图 4.3 用刚才讨论的例子描述了这个概念。

1 再次重申，applicative作用执行所有函数，不依赖其中任何一个函数的执行结果。

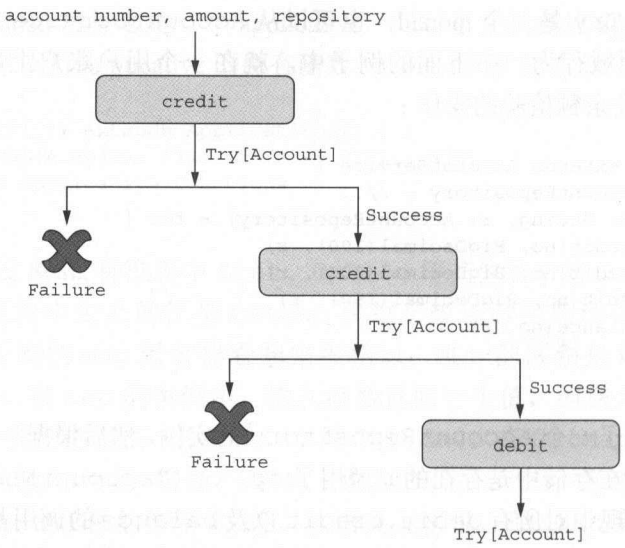


图 4.3 在单子计算中，计算过程依赖每一步的计算结果。如果函数从 Try 中返回的是 Failure，那么计算将可能在标记了 X 的任何地方终止。只有到每一步都返回 Success，整个计算才能完成。可以与图 4.2 的 applicative 计算做下比较。

正如我们所见，不管是 applicative 还是单子计算，在对领域进行建模时都有它们各自的作用。需要建立一系列独立的计算时，就会用到 applicative 作用；而需要能够快速失败而且是有条件地执行路径时，应该用单子作用。

领域模型设计中的单子与 applicative 作用

将 monad 或 applicative 的多作用编程模型加入到领域模型的实现中会给我们带来什么好处呢？两种计算结构都能帮助我们建立强有力的抽象，这样核心领域模型就可以保持干净简洁。下面是比较重要的几点：

- 基于附加结构也就是作用的函数应用的威力。我们已经了解普通函数是如何组合的，也知道如何将它们应用到一个参数。但这对领域模型来说还远远不够，需要在应用的某个点应用一个定制行为的函数。观察 flatMap 所提供的函数应用：def flatMap[A,B](ma:F[A])(f:A=>F[B]):F[B]。它将 A=>F[B] 应用到 F[A]，并将 F[F[B]] 变为 F[B]。通过对应用的平整使我们能够取回提供给 flatMap 的结构。这也确保我们可以基于 F[_] 的多作用结构来序列化函数应用。
- 基于附加结构也就是作用的函数组合的威力。如果有一个函数 f:A=>F[B]，还有另一个函数 g:B=>F[C]，这里的 F 是一个 monad，那么可以将它们组合为 A=>F[C]。这不是普通的函数组合，是单子函数的组合，也

被称为 *Kleisli* 组合。在 4.4 节中会有一个比较复杂的例子来演示它的威力。其中一个著名的例子就是 State monad，它允许组合多状态函数并自动地管理其中的状态。[1]

- 失败自动处理。面对函数应用或组合的序列时，需要考虑中间状态失败的处理。任何一个独立函数都有可能发生失败，同时需要通知到 API 的客户端。monad 和 applicative 将失败处理作为它们计算模型的组成部分，当然我们也看到了它们在处理失败方式上的差异。不过底线是保证核心领域逻辑免受错误检查的干扰。

Applicative 还是 Monad：应该用哪个

作为 Applicative 的子类型，Monad 是一个更加专用的抽象，它的功能更加强大，但 applicative 更加通用，也因此比 monad 应用得更广泛。在设计领域模型时，会发现在很多场景中，尽管我们一开始的想法是用 monad 来进行建模，但实际上 applicative 可能会更适合。这也是常说的，为使用场景选择威力最小的抽象。尽可能使用 applicative，如果需要 flatMap 再去使用 monad。

如在之前代码中定义的 Monad trait（Monad 的完整实现可参考 Scalaz[2]），可以为任何类型的构造器 F[_] 创建一个 monad，这样就可以有目的地实现代数。不过一样也可以通过扩展 Monad trait 的子类型来创建特定目的的 monad。因为 Scala 是一个对象函数式语言，我们既可以使用参数的多态性（通过 F[_] 的不同实例进行扩展），也可以使用子类型多态性（通过子类型进行扩展）。在下一个小节中将会讨论这样的一个特化——State monad——以及借助 State 来讨论如何有泛型单子组合器来实现领域模型的行为。到目前为止，我们还没有实现过组合器，接下来将使用 Scalaz 中的组合器并更多地聚焦在如何用它们来实现模型的特定行为。

State monad——在模型中管理多状态计算

关于函数式编程如何将值作为参数传递给纯函数我们已经讨论了很多。不过在现实生活中对领域进行建模时，需要频繁地管理状态，这些并没有传给函数。不过需要一直跟踪它们，从它们读取值，向它们写入值，把它们当成可变对象。多状态编程的一个典型案例就是随机数的生成，更准确地说，应该是伪随机数的生成。用一个初始值作为种子，它被用来生成一系列的数字，它们看起来就像自然的随机数一样。随机数生成器会维护一个全局状态，每生成一个新的数字它就会被更新。但

1 4.2.3节中也包含了State monad。

2 Scalaz中Monad的实现与这里所讨论的会有一点不同。它更接近于Haskell的定义。它为Bind定义了一个单独的抽象，然后将其与Applicative混合形成一个Monad。不过它们的基本概念是一致的。

当调用随机数生成器函数时，从来不会传入明确的状态。用 scala.util.Random.nextString(10) 这种方式进行调用，然后生成一个长度为 10 的 String 类型的值。

对于这种多状态计算的管理，一种方法是通过明确的可变数据结构。在 Scala 中，可以使用 var 来存储一个状态，并在需要改变状态时直接修改它。或者在函数式编程中，也可以选择在执行过程中携带状态，就像在递归调用中所做的那样。对于更好的抽象来说，携带状态是件沉重的事。State monad 可以提供抽象来保存状态过程，作为它计算结构的组成部分，同时它还利用 monad 序列化的能力（flatMap）来完成此项工作。作为该抽象的用户，我们要做的就是提供应用逻辑——如何修改状态，或如何从计算中提取值。于是再次传入纯函数。就像之前做的，结合我们领域中的例子，并用到 Scalaz 中实现的 State monad。

让我们考虑这样一个例子：我们已经获取了几个账户的余额列表，然后希望根据白天这些账户所发生的交易对余额进行更新。这是银行业务中批量处理完成的典型操作。当用户进行一笔交易时，后台维护的账户余额可能并不会立刻被更新。为了更容易阅读，下面的代码定义了一些类型别名，同时假设余额的列表已经被取出并存入到一个 Map 中，账户编号与余额一一对应：

```
type AccountNo = String
type Balances = Map[AccountNo, Balance]
val balances: Balances = …
```

这个 Map 就是需要管理的状态。当处理一个交易时，查询 Map 获取现存的余额，然后更新 Map。这里有个 Transaction 的简单模型：

白天发生的交易列表

Transaction 是白天账户交易的数目。负数表示这是一个 debit 交易，而正数表示这是一个 credit 操作。

```
case class Transaction(accountNo: AccountNo, amount: Money,
val txns: List[Transaction] = …
```

我们将使用 State monad 来管理这个余额的多状态计算。让我们从 Scalaz 中引入 State 模块以及提供组合器的相关对象，如清单 4.5 所示：

清单 4.5 使用 State monad 更新余额

使用本章代码中自己的 Monoid 实现。也可以用 Scalaz Monoid 使代码更加简洁。

```
import Monoid._
```

Scalaz 中的 State 模块。

引入的对象为我们提供组合器，马上就会用到。

```
import scalaz.State
import State._

def updateBalance(txns: List[Transaction]): State[Balances, Unit] =
  modify { (b: Balances) =>
    txns.foldLeft(b) { (a, txn) =>
      implicitly[Monoid[Balances]].op(a, Map(txn.accountNo ->
➥  Balance(txn.amount)))
    }
  }
```

下面列出了这段代码的重点，它在管理账户余额中使用了 monad 组合器：

- State 是一个 monad，它被定义为 State[S,A]，是个函数 (S)=>(A, S)，将一个状态 S 作为输入，然后输出一个状态 S 和值 A。在这个案例中，状态是 Map[AccountNo,Balance]。值可以是要提取出来作为计算组成部分的任何信息。在这个例子中并不需要任何值，只是想根据白天发生的交易更新状态。因此对于这种情况，我们采用了 State[Map[AccountNo,Balance],Unit]。
- modify 是定义在 State monad 上的组合器，如 def modify[S](f:S=>S):State[S,Unit]。它修改状态，并返回 monad 内更新后的状态。我们需要传递一个纯函数给 modify，通过它来更新并返回修改后的状态——这也正是自己携带状态并通过计算进行处理时所做的事。
- 在 updateBalance 中，用到了一个更新函数，它获取 Map 并根据输入的账户编号修改相应的余额。顺便说一句，Balance 是一个 monoid。[1] 因此可以将它用作 Map 中的值，使其也成为一个 monoid。我们可以借助 monoid 的力量使更新看起来更容易也更方便阅读——这也是适当抽象水平编程好处的另一个例子。

updateBalance 将返回什么？它返回 State monad：State[Balances,Unit]。所以我们还没有提供足够的输入给 monad 来运行计算，只是通过组合需要的抽象建立了计算。updateBalance 中没有我们提供的初始状态，命名为 balances 的 Map 来自所有计算开始的地方。我们只是表达了自己意愿：更新状态需要调用 modify，同时它需要将 Map 作为它的输入。函数式编程的组合能力也是其中一个原因，将抽象组合成小模块，然后根据需求再将不同部分黏合到一起。这样代码就拥有更好的可重用性和更好的模块性。

如果想要对一个交易列表运行 updateBalance，该 monad 有一个执行计算的返回方法，不过要给出一个状态的初始值：

1 关于Balance的Monoid定义，参见本章代码库。

```
updateBalance(txns) run balances
```

从领域建模的观点来看，State monad 给我们带来了什么呢？State 不光提供了前面章节讨论过的所有单子作用，还承载了模型的状态，因此在应用代码中就不用再去做这些事了。所有这些都是将管道逻辑指派给 monad 本身，同时完全用泛型的方式实现，这样就可以重用 State monad 和它的组合器来管理任意状态。作为一个领域建模者，只需要明确希望如何改变状态的业务逻辑，并将其定义为纯函数即可。monad 借助函数组合的力量，提供了一个包含一系列组合器且良好封装的抽象。

领域建模中的 State monad

我们需要在领域模型中管理变化的状态，在真实的模型中不可能所有事物都是无状态的。使用 State monad 可以使状态处理引用透明。在本节的例子中，updateBalance 就是引用透明的，同时还告诉我们如何通过 State monad 处理变更。

烦人的 monad 组合器——它们好在哪里

关于 State monad 的讨论指出，组合器 modify 接受一个纯函数并用它来更新模型的状态。不过 modify 是 State monad 所特有的，我们也讨论了可以用于所有 monad 的泛型组合器。本节会介绍其中一个，并用它实现模型中一个指定的特性，可以改造自己的问题去适应此类组合器的需求。一开始这可能不是很直观，特别是在从事命令式编程的情况下。不过一旦习惯用函数的方式思考之后，对这些组合器就会越来越有感觉。

在领域模型中，每个账户都有一个系统中唯一的账户编号。毫无疑问，模型中需要有生成这些编号的逻辑。不同的实现方式有不同的逻辑，但每种逻辑都将生成独一的编号。一旦生成一个编号，就需要检查它是否已经被分配给其他账户——这是个必要的检查，因为计算中可能会发生运算失败、数字溢出以及很多其他奇怪的事。总之，需要生成并检查直到获得一个唯一的账户编号。让我们来写一个这种生成器的抽象，它生成一个账户编号，同时包含一个函数来检查编号的唯一性：

```
final class Generator(rep: AccountRepository) {
  val no: String = scala.util.Random.nextString(10)
  def exists: Boolean = rep.query(no) match {
    case Success(Some(a)) => true
    case _ => false
  }
}
```

AccountRepository 是存放账户的存储。关于 Repository 的细节参见第 3 章。

生成逻辑将会很复杂。暂时先假设是个随机字符串。

查询存储检查唯一性。

如果就 State monad 这方面来考虑，Generator 类就是我们计算的状态。它生成一个账户编号，同时允许我们检查它的唯一性。如果这个编号已经存在且用于其他账户，就可以创建另外一个 Generator 实例来为账户获取新的编号。以下代码就是该逻辑的命令式的版本。状态 gen 会一直变化，直到合适的实例给我们一个唯一编号：

```
var gen = new Generator(r)
while (gen.exists) {
  gen = new Generator(r)
}
```

可以将 Generator 用作状态，同时将之前命令式的逻辑抽象成一个组合器，这样就有一个函数式接口开放给用户。在 Scalaz 中，这个组合器被称为 whileM_，其定义如下：

```
def whileM_[A](p: F[Boolean], body: => F[A]): F[Unit] = ..
```

其中的动作部分，也就是 body，会一直重复执行，直到 Boolean 表达式返回 true。我们需要的只是最终的状态，因为从中可以取得生成的账户编号，所以可以抛开计算结果不用管。组合器 whileM_ 借助纯函数的力量，将这种命令式逻辑转变为声明形式。使用组合器对这种命令式逻辑进行建模会非常直接易懂，如清单 4.6 所示：

清单 4.6 使用单子组合器生成有效的账户编号

```
val StateGen = StateT.stateMonad[Generator]
import StateGen._
val r: AccountRepository = //..
val s = whileM_(gets(_.exists), modify(_ => new Generator(r)))
val start = new Generator(r)
s exec start
```

趁着 State monad 的知识还在脑海中盘旋，这个组合器还是非常直观的。不久我们将消除细节方面的问题。但最重要的一点是，我们依然在处理纯函数以及它

们的组合性。接下来剩余的部分将解释 whileM_ 是如何完成我们所期望的内容的：

- StateT 是一个 monad 转换器。它在这里所做的事情不是非常重要，除了 StateT 的 stateMonad 函数会生成一个 Generator 作为状态的 State monad。
- 第二行的 import 使得 State monad 内部的组合器以及泛型 Monad 抽象都可用。（注意，State 同样是个 monad。）
- modify 是我们之前使用的组合器，它获取一个函数来修改状态。在我们的场景中，它创建了 Generator 的新实例来生成另一个账户编号。
- gets 应用传递给它的转换函数并返回更新后的 State monad。在这里它会检查生成的账户编号是否已经存在。只要 exists 返回 true，就要继续执行 whileM_。
- 如果希望执行计算并生成下一个有效的账户编号，需要传入一个 Generator 的启动配置并在 monad 上触发 exec。

whileM_ 是一个可以为所有 monad 工作的组合器，之前所讨论的例子就是告诉大家如何为领域模型用泛型抽象来搭积木。识别使用场景以及相应的组合器，并提供它所需要的函数。然后组合器就会在幕后帮我们把所有事情都办了。

4.3 如何用模式对领域模型进行塑形

到目前为止，我们已经看了不少的模式。除了函数组合的基本技术，还了解了注入 monoid 这样的模式，用来帮助我们在联合二元操作上组合代数结构。在本章中，将看到多作用编程的模式——作用的两种类型，它们将对计算的结构进行塑形。我们已经看到它们如何被运用于领域模型的上下文中，但这些模式能发挥多大的作用？是不是几乎不会用到它们？还是它们会被非常频繁地使用，以至于在一本领域建模的书中需要用大量篇幅来讨论它们？

前面的章节反复重申，函数式编程的本质在于纯函数的能力：在混合上增加静态类型，就拥了有代数抽象——在类型上操作的函数，且遵守某些法则；使函数在类型上泛化，就拥有了参数属性；函数变成多态的之后，就具备了更高的可重用性，如果我们训练有素，在特化类型（或不受管理的危害，如异常）中就不会泄露任何实现细节，这样就得到了免费的定理。[1] 这就是我们所说的，函数式抽象或模式所拥有的精髓的阶梯。图 4.4 展现了这个精髓的阶梯，当我们脑子里环绕着各种函数式编程的细微区别时，它可以帮我们逐步理清思路。

1 当我们有一个多态函数时，会自动获得一些基于函数类型的定理。参数属性会给我们提供一些免费的定理。更多细节请访问http://homepages.inf.ed.ac.uk/wadler/topics/parametricity.html查看Phil Wadler的论文*Theorems for Free*。

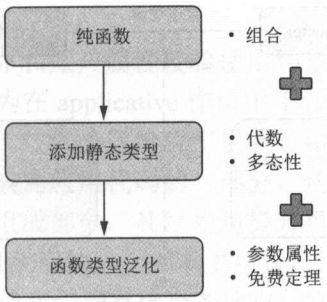


图 4.4 精髓阶梯。当拥有纯函数时，可以大胆地组合。添加静态类型，获取代数抽象。当这些函数在类型参数上彻底泛化时，即可获得免费的定理。

领域建模中的模式精髓 #1 当模式在类型上泛化并参数化时，通常就非常有用。在这种情况下，能够建立可用于任何符合抽象代数的类型的结构。当我们有一个泛型模块 `Monad[F[_]]` 时，也就可以为任意类型的构造器定制实现（比如 `List` 和 `Option`）。

结合泛型的优势，模式使我们可以定义泛型组合器，这也正是我们在 monad 和 applicative 中所看到的。这些组合器只需要实现一次，然后就可用于所有对该模式实现的特化。比如之前看到的例子，在 `Monad` 模块中定义的 `whileM_` 组合器，以及为 `State` 定义的 `modify` 和 `gets` 组合器。Don Stewart 是这样描述的："命名一次，实现一次，无限重用。"[1]

领域建模中的模式精髓 #2 本质上，泛型模式提供的泛型组合器和函数适用于模式的所有特定实例。一个 `Applicative` 模块上定义的组合器 `sequence`[2] 可被用于 `Applicative` 的所有实例（比如 `Applicative[List]` 和 `Applicative[Option]`）。

让我们更深入地了解这些模式给领域模型带来的架构方面的价值。回顾一下 4.1.2 节中实现的 applicative 作用，我们用它在创建账户之前校验账户属性（图 4.2 描述了这个概念）。如果没有 Applicative Functor 模式，代码就会变得非常的冗长，同时，应用代码还不得不处理很多偶发的复杂事件，比如异常处理、所有属性的检验链，以及基于结果的有条件执行。这些都会使代码变得非常非常复杂，真实的领域逻辑将混杂在这些附加的关注点里。图 4.5 清晰地描述了泛型 applicative functor 抽象和领域逻辑分别要处理的关注点。

1 参见Don Stewart的*Haskell in the Large*（http://web.archive.org/web/20150129012424/http://code .haskell.org/~dons/talks/dons-google-2015-01-27.pdf）。

2 `sequence`定义在Scalaz的类型类`Traverse`中，参见https://github.com/scalaz/scalaz/blob/series/7.2.x/core/src/main/scala/scalaz/Traverse.scala。

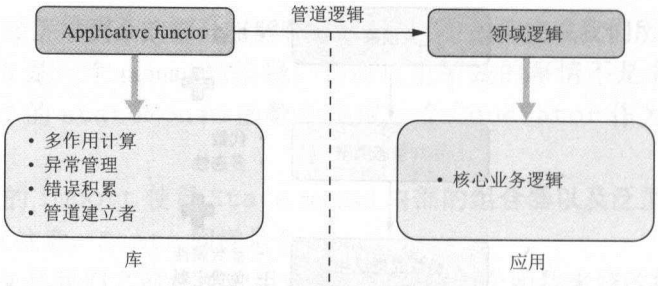


图 4.5 模式是泛型结构，通过管道代码，能够被黏合到领域模型上。要注意一个好的设计模式自身所能处理的关注点的数量。这会占用很大一部分应用的代码，在没有附带的复杂事件时，这部分会变得更加简洁并且有更好的表达性。

领域建模中的模式精髓 #3 模式将抽象的公共部分抽象出来，用户仅需要实现领域特定的易变部分。这使应用代码简洁明了，更容易理解，且没有任何附带的复杂性。

设计模式可以组合——至少它们应该这样做。我们应该学会组合多个模式以获取更高水平的模式。让我们看看 Scalaz 中的 `Validation` 抽象。它在用法上和 4.1.2 节所讨论的非常类似，但它有更广泛的组合器、与其他抽象的互动性、以及其他很多特性，这些使它成为一个可直接应用于执行领域对象校验的结构。以下是 Scalaz 中 `Validation` 的基本结构：

```
sealed abstract class Validation[+E, +A] { //.. }
final case class Success[A](a: A) extends Validation[Nothing, A]
final case class Failure[E](e: E) extends Validation[E, Nothing]
```

这看起来和 `scala.Either[+A, +B]` 极度相似，它在 `Left` 和 `Right` 也有两个变量。事实上，`scalaz.Validation` 和 `scala.Either` 是同构的。[1] 那么在这个情况下，为什么 `scalaz.Validation` 是一个单独的抽象呢？`Validation` 使我们可以积累失败，这在设计领域模型时是一个很常见的需求。一个典型场景就是当我们在校验一个基于 web 且包含很多字段的表单时，需要一次性将所有错误返回给用户。如果用包含 `Semigroup`[2] 的 `Failure` 类型构造一个 `Validation`，那么库将为 `Validation` 提供一个 applicative functor，它可以在计算过程中积累所有错误。这同样也是使用诸如 Scalaz 这样的库的重要动机：可以享受在 Scala 标准库之上更强大的抽象。`Validation` 就是这些函数式模式中的一个，它使代码更加强

1 直观上来看，同构比相等具有更宽松的关系。相等表示两个抽象必须完全相同，而同构是说两个抽象可以互相转换。在当前的情况下，给出`Validation[E, A]`，可以构建一个相等的`Either [E, A]`，反过来也一样。

2 Semigroup是一个没有0的monoid。

大简洁。

在关于 applicative functor 和账户属性校验使用场景的讨论中，并没有谈论太多有关失败处理的策略。但因为在 applicative 作用中不管它们的结果是什么，都将独立执行所有校验函数，所以比较好的策略就积累所有错误并一次性报告给用户。但问题是，错误处理策略应该成为应用代码的一部分，还是应将其抽象在模式中？将错误处理策略抽象成模式的组成部分，其好处是提升可重用性，减少应用代码中的样板代码，这也是 FP 所吸引我们的。正如本书所说，一个漂亮的 Applicative Functor 和 Semigroup 模式的组合使我们拥有这个关注点，它被抽象在库里。如果使用这种方式，最终将用函数式模式来组合代码。通过构造，类型将在确保抽象的正确性中发挥重要作用，同时实现细节也不会泄露到特定应用的代码中。在练习 4.2、4.3 以及本章代码库中将会有其他一些例子来探索更多相关内容。

练习 4.2 积累校验错误（Applicative）

4.2.2 节中提供了 Applicative Functor 模式，并用它对领域实体的校验进行了建模。观察下面的函数，输入一系列的参数，返回一个全面构造的有效的 `Account` 给用户：

```
sealed abstract class Validation[+E, +A] { //.. }
final case class Success[A](a: A) extends Validation[Nothing, A]
final case class Failure[E](e: E) extends Validation[E, Nothing]
```

- 函数的返回类型为 `scalaz.ValidationNel[String, Account]`，它是 `Validation[NonEmptyList[String], Account]` 的简化写法。如果所有校验都成功，函数返回 `Success[Account]`，否则它必须在 `Failure` 中返回所有的校验错误。这也就意味着所有校验函数都要运行，不管运行结果如何。这就是 applicative 作用。
- 我们需要实现如下校验规则：（1）账户编号的最小长度是 10 个字符；（2）利率必须为正；（3）开户日期（若未特殊说明，默认为今天）必须早于关闭日期。
- 提示：仔细看一下 Scalaz 中 `Validation[E, A]` 的实现。留意它是如何提供一个 `Applicative` 实例通过 `Semigroup[E]` 来支持错误消息的积累。要注意的是 `Semigroup` 是不包含 0 的 `Monoid`。
- 提示：3.2.2 节关于智能构造器的讨论中有个相同函数的定义，代码库中有它的实现，可以从这里开始。

领域建模中的模式精髓 #4 模式组合。可以将多个小模式组合成一个大模式。

我们现在已经知道如何使用设计模式建立更好的、在模型中可重用的抽象来提升领域模型，让我们来看几个字段的故事。有一些使用函数式编程的领域模型的使用场景，它们用类型、代数和模式的组合来获得更好的抽象、模块化及可维护性。4.4

节中提供了一个 API 演进的完整案例，从代数开始，借助类型、模式和函数组合的力量，在抽象上不断做增量建设。在做任意特定实现之前还有很多事可以做。

练习 4.3 快速失败的校验（单子）

4.2.3 节中讲到了单子作用，并将它们与 applicative 做了比较。在练习 4.2 中，学习了处理校验的 applicative 方式。在这个练习中，将学习作用的单子应用。让我们借用练习 4.2 里一样的函数，但要实现账号属性的快速失败校验：

```
def savingsAccount(no: String, name: String, rate: BigDecimal,
  openDate: Option[Date], closeDate: Option[Date],
  balance: Balance): ??? = { //..
```

- 函数的返回类型是需要我们做决定的事情。这里的想法是返回一个 monad，在执行后如果都成功就返回一个构造的 `Account`，否则就返回一个失败，其中包含了第一个校验失败的信息。提示：Scalaz 有一个很好的抽象来做这件事。
- 我们需要实现以下校验规则：（1）账户编号的最小长度是 10 个字符；（2）利率必须为正；（3）开户日期（若未特殊说明，默认为今天）必须早于关闭日期。
- 每个校验都实现为一个独立函数，然后在主函数 `savingsAccount` 中用 for 表达式将它们组合在一起。

4.4 用代数、类型和模式演进API

第 3 章中已经展现了如何用领域期望功能的代数演进 API。在本节中，会看到更多有趣的案例，用同样的代数方式去发现函数式编程现有的模式如何被很好地运用于组合领域行为，并引导 API 设计。我们将从一个典型的使用场景开始，思考支持该场景的代数，抽象出类型，然后试着把它们组合到一起。这将是个有趣的练习，结果也会非常具有说服力，我们会见识到使用现有模式函数组合的威力。但首先需要对讨论的领域做一些提升。要从个人银行领域中走出来，将范围扩大到投资银行的业务领域。我们不再讨论 debit 和 credit，而将谈到客户单据、成交、交易以及结算。通常来说，很多提供个人银行业务的金融机构同时也拥有独立的投资银行业务，它主要服务于机构与零售客户。下面的贴士中描述了投资银行业务的几个基本概念，在后面的例子中会用到它们。

证券业务基本概念：单据、交易、成交

在我们的例子中投资银行业务的特定领域与交易业务和证券兑换相关（就像我们熟知的证券市场）。下面有一些需要知道的基本概念，以便更好地理解本节中的例子：

- 就像在银行开设一个个人业务账户一样，也可以在一个投资银行开设一个交易账户或结算账户。如果开设一个交易账户，可以通过它进行证券交易（股票和债券）。交易既可以是买入，也可以是卖出。通过结算账户对交易进行结算。
- 如果是交易账户的用户，可以通过银行交易订单。一张单据包括要交易（买入或卖出）的证券名称（或ISIN）、要交易的数量、价格范围，以及一些在当前上下文中非必要的信息。
- 当客户的单据到达交易组织（投资银行）之后，它们会被放到兑换（市场）等待成交。
- 市场完成交易并将成交结果返回给银行。它们并不会直接对应到每个单独用户所下的订单上。这些成交结果会匹配到某一天所有用户所下的所有单据上。
- 银行在收到成交结果之后将进行分配，通过交易的方式将成交结果对应到不同账户中。这些交易将与客户在银行中所下的订单保持对应。

我们的案例场景将贯穿以上过程，因此也没有必要再深入了解更多的细节。当然，如果你感兴趣，Investopedia（http://investopedia.com）提供了更多关于投资银行业务的细节内容。

在我们的例子中，用户场景是这样的：

- 客户以特定的格式给交易员下单。
- 交易员将其转换成内部格式（用于客户订单的领域实体），并在市场中下单。
- 当交易在市场中完成之后，交易代理取回成交结果。要注意，成交结果并不会对应到客户订单。所有的客户订单被聚合在一起，然后被放到市场中进行交易。因此聚合了一堆客户订单后将这个篮子放到市场中等待成交。
- 交易代理取回成交结果之后，它们将以交易的形式分配到客户账户，同时各账户的余额也会得到更新。

先不考虑任何实现的细节，让我们先来假设一些领域实体的类型以及值对象，并试着创建函数的第一版草稿。

4.4.1 代数——第一稿

领域 API 设计的第一原则是坚持领域的业务语言。我们按照这个原则完成 API 设计的第一版，如清单 4.7 所示。

清单 4.7 Trading API 的代数

```
trait Trading[Account, Market, Order, ClientOrder, Execution, Trade] {
  def clientOrders: ClientOrder => List[Order]
  def execute: Market => Account => Order => List[Execution]
  def allocate: List[Account] => Execution => List[Trade]
}
```

在模块中定义的这些函数根据模型中的行为进行了命名。我们只是简单地抽象了一些类型，还没有考虑它们的实现。模块 Trading 根据这些类型进行参数化。当我们在明确代数时，也别想着讨论它们的实现——代数是我们需要注意的解释。通过前面的说明，以及所拥有的领域知识，我们定义了 Market、Order、Execution、Trade、Account 和 ClientOrder 作为领域实体的例子。API 的类型签名同样包含了领域的某些规则(比如，一个 ClientOrder 可以指向多个 Order 实例)。

如果仔细观察这 3 个函数，就会发现我们需要采用明确的步骤来服务 4.4 节中所描绘的场景。图 4.6 展示了该场景的步骤，其中使用了之前定义的 3 个函数。

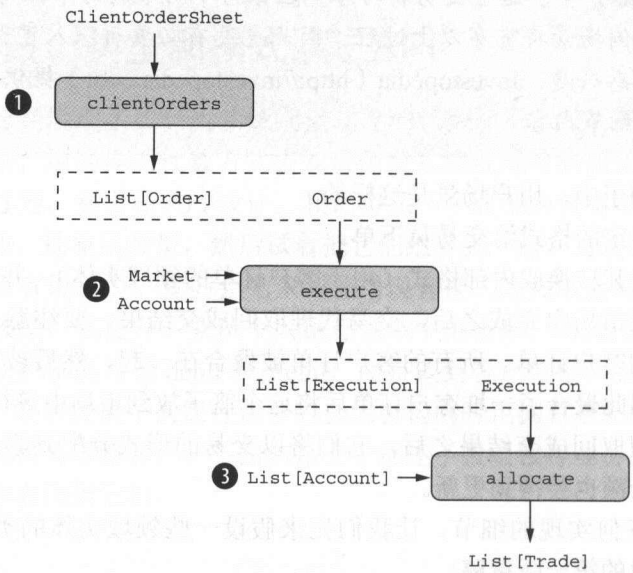


图 4.6 通过文中定义的函数对 Trading 的步骤进行建模。图中显示了函数的输入和输出——关于如何将它们组合到一起用于场景中的连续执行，参见文中的详细内容。

4.4.2 改进代数

清单4.7中所定义的函数的代数，并不能很清晰地通过它们达到图4.6所期望的序列。图中的虚线方框给出了一个暗示：clientOrders输出一个List[Order]，而execute需要一个Order。让我们重新组织下函数的参数，然后再来看结果如何。

清单4.8 Trading API的代数（轻度重构）

```
trait Trading[Account, Market, Order, ClientOrder, Execution, Trade] {
  def clientOrders: ClientOrder => List[Order]
  def execute(m: Market, a: Account): Order => List[Execution]
  def allocate(as: List[Account]): Execution => List[Trade]
}
```

概括地说就是有3个A=>M[B]、B=>M[C]以及C=>M[D]的函数，同时我们的期望是将它们组合到一起。那么大家了解M吗？在当前的例子中，M是一个List，同时List是一个作用，就如之前看到过的（第2章）。但在函数式编程中，我们通常会试着归纳计算结构，这样就可以得到一些模式并在广泛的上下文中进行重用。因此在一个更泛化的作用上组合这3个函数，也就是要用Monad而不是List。可以通过一种被称为*Kleisli*箭头（arrow）[1]的代数结构来做到这一点，它允许函数f: A => M[B]和g: B => M[C]进行组合并生成A=>M[C]，在这里M是一个Monad。这看上去就像普通的组合，只不过是基于多用作的函数。

Kleisli只是函数A => M[B]上的一个包装。如果作用（类型构造器）是一个monad，就可以组合两个Kleisli。以下是Scalaz中Kleisli的一部分定义（简化版）：

```
case class Kleisli[M[_], A, B](run: A => M[B]) {

  def andThen[C](k: Kleisli[M, B, C])(implicit b: Monad[M]) =
    Kleisli[M, A, C]((a: A) => b.bind(this(a))(k.run))

  def compose[C](k: Kleisli[M, C, A])(implicit b: Monad[M]) =
    k andThen this

  //..
}
```

Kleisli只是对多作用函数进行了包装。正如在3.2.1节中所讨论的，一个计算建立一个没有赋值的抽象。Kleisli就是这样的计算，它不会做任何赋值，直到调用函数运行。可以通过这种方式建立多个级别的组合，而不用过早地进行赋值。在本节中用这个特性建立了交易的API。

1 术语箭头（arrow）来源于范畴理论；在Scala或Haskell中，它对应到一个函数。

使用 Kleisli 组合，将 3 个函数组合成包含作用的函数。同时这也演示了如何用现成的代数结构表示领域代数。在讨论代数 API 设计时，这是非常重要的一个概念。清单 4.9 展示了如何用刚才讨论的新模式来改进领域代数。

清单 4.9 Trading API 的代数（使用 Kleisli 模式）

```
trait Trading[Account, Market, Order, ClientOrder, Execution, Trade] {
  def clientOrders: ClientOrder => List[Order]
  def execute(m: Market, a: Account): Order => List[Execution]
  def allocate(as: List[Account]): Execution => List[Trade]
}
```

使用 Kleisli 模式之前

```
trait Trading[Account, Market, Order, ClientOrder, Execution, Trade] {
  def clientOrders: Kleisli[List, ClientOrder, Order]
  def execute(m: Market, a: Account): Kleisli[List, Order, Execution]
  def allocate(as: List[Account]): Kleisli[List, Execution, Trade]
}
```

正在使用 Kleisli 模式

4.4.3 最终组合——采用类型

现在我们已经掌握了如何用 3 个函数来建立一个更大的交易的领域行为。只要采用类型，就可以拥有完整的交易处理（当然是仅用于演示的简化版）——从客户订单到交易分配进客户账户。

清单 4.10 客户订单的交易处理

```
def tradeGeneration(
  market: Market,
  broker: Account,
  clientAccounts: List[Account]
) = {
  clientOrders andThen
    execute(market, broker) andThen
      allocate(clientAccounts)
}
```

好好观察这个领域行为的实现，它根据客户订单产生交易。若将它和 4.4 节中列出的业务规则做比较，就会发现实现紧紧地跟随着规则，而我们也获得了通用的业务语言。这个例子再一次演示了我们可以在较低的水平组合函数，在较高的水平组合模式，然后得到一个表达清晰的领域模型的实现。

在领域驱动设计中，Eric Evans 认为一个好的领域模型应该具备以下一些特质：

- 利用普遍的业务语言拟定能自我阐述目的的接口。
- 将隐形的概念明确化。
- 无副作用的函数。
- 陈述性设计。

大家可以在我们的方法中看到所有这些特质，它使用静态类型的函数式设计模式作为模型实现的基础。

练习 4.4 解密 Kleisli

清单 4.10 中建模了完整的从客户订单开始的交易处理。它借助 Kleisli 代数组合了 3 个多作用函数。Kleisli 给了我们一个计算（我们结合清单 4.9 中的代码对此开展了讨论），对此需要调用底层函数来赋值。

- 为一个市场、一个经纪人账户以及一个客户账户列表运行之前的 `tradeGeneration` 函数。计算的结果和类型是什么？可以从本书代码库中找到支持的抽象（市场、账户等）。
- 在上一步中，有没有获取到最后的交易列表作为输出？要确定自己获取的结果能作为输出。
- 还能额外做些什么来生成交易列表的最终输出？

4.5 用模式和类型增强领域的不变性

对一个领域进行建模时，行为需要遵守业务的相关约束。有很多种方式来保证这些约束：可以编写纯业务逻辑，并在运行时行为中执行它们，这也是使用动态类型语言时最常用的技术。如果有一个静态类型系统，就有了很多额外的选项。类型在很多领域约束的编码中具有极大的作用。而且在能够完成此项内容时，还能同时获得很多免费的东西。比如测试，编译器会替我们完成，还获得了参数属性，这比写任何形式的测试用例更加管用。本章节提供另外一种使用模式的函数和类型的组合，它可以用来为领域模型对领域不变体进行编码。

4.5.1 贷款处理模型

银行业务模型需要处理用户的贷款申请。银行需要校验申请，通过适当的授权使其得到批准，若匹配条件则处理贷款，最后发放贷款。这是非常明确的工作流，同时该工作流在处理贷款申请的过程中有一系列模型必须遵守的约束。

让我们来看看清单 4.11 的贷款申请模型。

清单 4.11 贷款申请模型

```
case class LoanApplication private[Loans](
  date: Date,
  name: String,
  purpose: String,
  repayIn: Int,
  actualRepaymentYears: Option[Int] = None,

  startDate: Option[Date] = None,
  loanNo: Option[String] = None,
  emi: Option[BigDecimal] = None
)
```

将构造器设置为私有，将会有替换的实例化方式。

贷款申请日期。

用户希望占用期限。

实际批准的期限（后面强化）。

贷款实际开始日期（后面完善）。

贷款账户编号（后面完善）。

EMI（后面完善）。

工作流中包含了明确的步骤，思考以下 3 个步骤所描述的目的：

- 接受贷款申请（用户）。
- 批准贷款申请（银行业务授权）。
- 补充完善批准的贷款申请，就是在前面模型中空出来的细节部分。

相应的，下面的内容对工作流进行了建模。前面的 3 个步骤需要被序列化。正如4.4节中所学到的，Kleisli是用来对多作用函数序列进行建模的有效手段。清单4.12展示了工作流建模的第一稿。

清单 4.12 贷款处理工作流函数的第一稿

```
def applyLoan(name: String, purpose: String, repayIn: Int,
  date: Date): LoanApplication

def approve: Kleisli[Option, LoanApplication, LoanApplication]

def enrich: Kleisli[Option, LoanApplication, LoanApplication]
```

applyLoan 函数获取相关的参数并构造了一个 LoanApplication 实例。approve 执行批准的处理：因为贷款申请有可能被拒绝，所以返回一个 Option 类型。如果贷款申请被批准，则 approve 函数填写模型的 loanNo、actualRepaymentYears 和 startDate 字段。enrich 计算 EMI 并完成模型。下面是如何使用该 API：

```
val l = applyLoan("John B Rich", "House Building", 10, today)
val op = approve andThen enrich
op run l
```

applyLoan 函数是智能构造器，它返回模型的一个实例。接下来就是使用 Kleisli 模式将工作流函数按正确的顺序串起来，然后处理贷款申请。

这看起来非常地整洁，将功能都包裹在 Kleisli 的细节中。当设计一个 API 时，其中一个目标就是要使它尽可能地健壮。或者应该使其足够安全，这样它就不会轻易地被错误使用，也不需要编译器过多地干预进来。对于我们写的 enrich andThen approve，思考一下 API 在之前的使用中会发生什么。对 Kleisli 来说类型是匹配的，编译器也很开心。但是，这样做违反了领域行为的不变性：我们不能在没有批准贷款申请的情况下去补充它。我们的运行时行为是不正确的，这种行为绕过了我们的编译器。

我们不可能通过类型系统找出所有的违反行为，但在这个案例中可以通过使用类型系统得到了很好的效果，办法就是通过更多类型来强制执行工作流的序列。在批准之前补充完善信息在工作流中是非法的状态，需要防止它的发生。

4.5.2 使非法状态不可表示[1]

在处理中使用了强大的类型系统时，不要害怕使用更多的类型。而且这也正是本例中所要做的。我们将使用额外的类型来使工作流状态更加明确。但这些类型只会发挥标记的作用，它们使状态变成唯一的，但在领域逻辑中并不发挥作用。我们用独立的类型来描述工作流的状态，然后得到用该类型参数化的模型 LoanApplication。清单 4.13 展示了修改后的模型。

清单 4.13 Phantom 类型的贷款处理工作流

```
case class LoanApplication[Status] private[Loans]( //..

trait Applied
trait Approved
trait Enriched

type LoanApplied = LoanApplication[Applied]
type LoanApproved = LoanApplication[Approved]
type LoanEnriched = LoanApplication[Enriched]

def applyLoan(name: String, purpose: String, repayIn: Int,
  date: Date = today) =
  LoanApplication[Applied](date, name, purpose, repayIn)

def approve = Kleisli[Option, LoanApplied, LoanApproved] { l =>
  l.copy(
```

LoanApplication 已经用类型进行了参数化。

Phantom 类型。

1 该模式的思路以及本节的标题均来自于Yaron Minsky一篇精彩的博客，它对OCaml语言中的影子类型展开了讨论（https://blogs.janestreet.com/effective-ml-revisited/）。

```
    loanNo = scala.util.Random.nextString(10).some,
    actualRepaymentYears = 15.some,
    startDate = today.some
  ).some.map(identity[LoanApproved])
}

def enrich = Kleisli[Option, LoanApproved, LoanEnriched] { l =>
  val x = for {
    y <- l.actualRepaymentYears
    s <- l.startDate
  } yield (y, s)
  l.copy(emi = x.map { case (y, s) =>
    calculateEMI(y, s) }).some.map(identity[LoanEnriched])
}

private def calculateEMI(tenure: Int, startDate: Date): BigDecimal = ..
```

清单中所引入的额外类型 Applied、Approved 以及 Enriched 并不参与任何领域行为的实现。它们只用来消除工作流状态的歧义，并强制执行以下约束：

- 一个新的 LoanApplication 总是处于 Appied 状态。
- 在处于 Applied 状态的 LoanApplication 上，唯一能调用的函数是 approve，同时它会将 LoanApplication 改为 Approved 状态。
- 只有在 Approved 状态时，才可以完善 LoanApplication，并将其变为 Enriched 状态。

通过静态的方式完成这些内容，并通过编译器得到强制执行。我们只能 approve andThen enrich。编译器如果遇到不正确的序列就会大喊大叫。由于这些类型只用于状态的明确，所以它们被称为影子类型（phantom types）。可以使用 Kleisli 模式将影子类型与 API 的纯函数式实现组合起来实现一个 API，从而静态地保证了所有领域的不变性。

影子类型模式可以有效地用来使领域模型的非法状态不可表示。这个模式与函数没什么关系。我们之所以讨论它，是因为它可以与函数式模式良好地配合工作，同时还能增强模型的健壮性。

4.6 总结

本章包含了一系列类型函数式编程的高级专题。建议大家反复阅读并充分理解我们所讨论的内容，同时也可以参考代码库中的代码。本章只谈到少量的模式，还有很多其他的模式可以用来改善模型。本章的主要内容如下。

- 什么是函数式设计模式？从函数式编程的视角所看到的设计模式与面向对象编程中的设计模式是有区别的。在 FP 中，模式是抽象的代数，可以根据上下文为它们编写特定的解释程序。

- 无处不在的 monoid：monoid 是一种设计模式，它经常出现在领域模型中。monoid 可以帮助泛化模型，对领域操作进行抽象。
- 用于多作用编程的模式：在类型函数式编程中 3 种最常见的模式就是 functor、applicative 和 monad。可以用它们在领域模型中构建计算。
- 模式的精髓：我们已经学过了模式精髓的内容，也看到如何小心应用函数式模式，进而使领域模型具备组合性、参数性以及可重用性。
- #1 实战：学习了如何借助类型和函数改进 API。还学习了如何结合 Kleisli 多作用函数组合来设计领域行为的工作流。
- #2 实战：学习了如何用影子类型来静态地保证领域不变性。还上了一堂关于类型系统的课，同时学习了如何用它来编码逻辑，使其可以被编译器自动测试。

5 领域模型的模块化

本章包括

- 将领域模型模块化
- 将领域模型拆分成模块的详细案例学习
- 理解模块如何聚合成边界上下文
- 使用模块的高级模式——free monad

对于一个大的模型，从整体中较小单元方面开始思考，可以帮助我们更好地理解它。这也是认知能力工作的方式。不要试图一下子理解整个银行业务系统的所有细节是如何工作的，应该首先去理解比较小的部分，比如客户账户管理、投资组合管理、报告服务以及后台模型。还有一些相对独立的子系统使用着不同的业务语言，它们可能会有单独的数据和领域模型。就算在每个子系统内部，同样也会有一些非常复杂的功能，难以作为一个整体去理解。

当本书谈到模块化时，是指将大的模型拆解到小型的模块中，它们是一个语义相关行为的组合。每个模块都有代数，它们被发布给客户，一个或多个实现都被仔细地保护着，以免因为粗心大意而导致与客户端代码的耦合。本章包含了领域模型的模块化，以及如何使用抽象来将模型拆分成可组合的模块。

图 5.1 展示了本章的几大专题。大家可以选择性地学习本章内容。

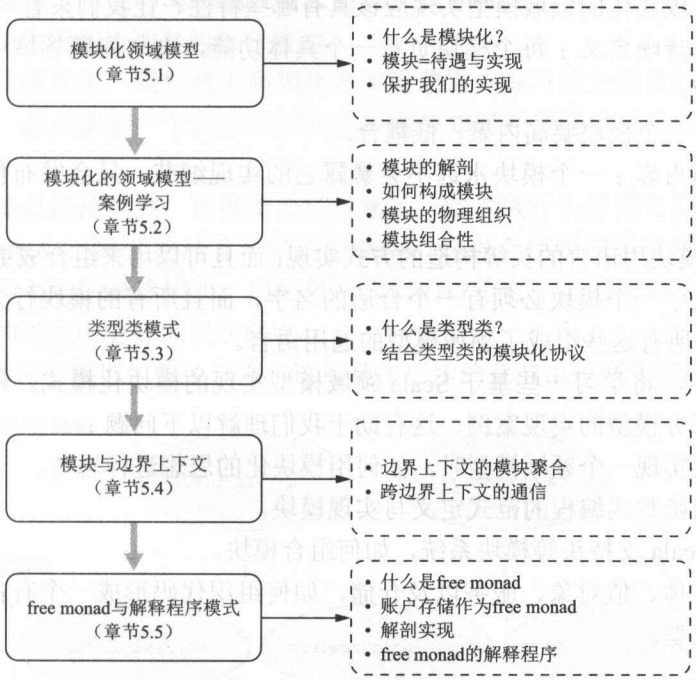


图 5.1 本章学习过程。

到本章结束，我们将理解如何使用合适的抽象来模块化领域模型。

5.1 将领域模型模块化

如果一个复杂系统被设计成一个庞大的泥球，那么这个系统通常是很难理解的。作为人类，我们总是试着从整体中先找到某些相对简单的部分，这些部分被组合在一起形成整个系统。这是我们大脑的工作方式，也是现实生活中认知能力映射到具体事务的方式。将整个系统分解成一系列简单部分的集合，就是模块化的本质内容。而那些简单的部分被称为模块。

在识别出系统中的模块之后，就会希望每个模块各自履行一个特定的职责。在个人银行领域的案例中，可能会有一个叫作账户服务的模块，它包含了所有管理用户账户的函数。为了实现这个功能，该模块的所有元素需要尽可能地彼此协同合作，最小化对其他模块的依赖。这使得模块内部具有很强的聚合性，同时最小化与其他模块的耦合。模块化不仅仅是为了更方便理解，它也是软件工程中重要的基本组成。一个拥有良好模块化的系统会更容易被理解、管理、重构以及测试，与那种一整块

的代码相比，模块化可以在整个软件生命周期开发过程中给出更清晰的思路。

一个良好模块化的领域模型实现应该具有哪些特性？让我们来看一下。

- 具体且清晰定义：每个模块负责一个具体功能，它与问题领域中的类似行为相对应。
- 内聚：一个模块要高内聚，低耦合。
- 发布的内容：一个模块永远不会暴露它的实现细节，只会发布代数给它的客户。
- 头等：模块用语言的头等构造的方式实现，而且可以用来组合成更大的模块。[1]
- 词汇表：一个模块必须有一个合适的名字，而且所有的模块行为也必须有名字——所有这些组成了领域模型的通用语言。

在本章中，将学习一些基于 Scala 领域模型实现的模块化模式。还将学习个人银行领域中部分模型的实现案例。这有助于我们理解以下问题：

- 在需要实现一个领域模型时，如何用模块化的思想进行思考。
- 如何用函数式编程的范式定义与实现模块。
- 基于 Scala 支持头等模块系统，如何组合模块。
- 基于实体、值对象、服务以及仓储，如何组织代码形成一个有良好架构的模块化模型。

5.2 模块化的领域模型——案例学习

让我们通过个人银行领域里的一个功能子集来描述模块建模的不同方面：

- 领域元素，实体与值对象。
  - 账户，该实体对客户账户进行建模。
  - 余额，该值对象对客户账户的余额进行建模。余额应该包含数额部分以及货币部分。为了简单起见，我们只考虑数额部分。
  - 数额，该值对象对余额的金额部分进行建模。
- 领域服务，对业务用例进行建模。
  - 账户服务（Account Service），实现处理客户账户的所有功能。它与我们在第 3 章和第 4 章中所讨论的领域服务 `AccountService` 非常类似。
  - 利息计算（Interest Calculation），它实现计算用户账户利息的逻辑（简化版）。现实中它包含了非常复杂的计算逻辑，在这里我们进行了简化。

1 不是所有语言都支持头等模块。支持头等模块的语言相对来说更有优势一些。Scala就提供了头等模块的支持。

- 所得税计算（Tax Calculation），它负责计算账户余额利息的所得税。

我们将使用组合之前的领域服务来实现某些领域行为，这将向我们展现一个设计良好的模块化模型所拥有的组合能力。

对所有这些元素，我们将采用简化的实现逻辑。练习的主要目的是讨论良好模块化的优点，以及演示一个良好的模块化实现是如何使代码更容易被理解、维护及重构的。

我们可能已经注意到，该模型已经识别了一些领域行为并把它们列为不同的功能（服务）。我们已经开始涉及小型的功能块，这也会使我们更容易理解。我们并没有谈到该模型中的最终模块。但如果对我们的讨论有一定的认识基础之后，将有助于在后面的进程中更好地识别模块。

图 5.2 展示了模型的整体范围，这也是在后续的章节中需要实现的内容。

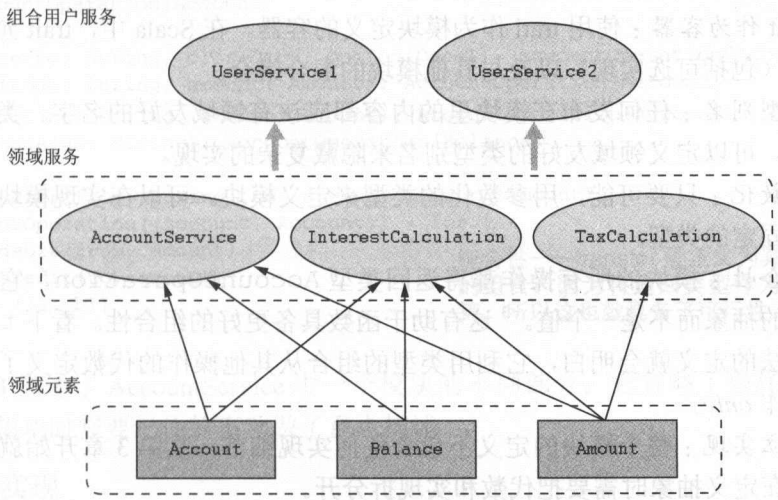


图 5.2 领域模型的范围。领域元素为实体和值对象。如箭头所示，领域服务使用元素。最后，通过组合领域服务来实现更高级的领域行为。

5.2.1 模块的解剖

之前列出的领域服务组成了领域的一连贯功能。它们为客户提供了实现特定用例所必须的完整函数集。我们会使它们每个都成为一个单独的模块，因为它们都满足 5.1 节中列举的模块的标准。

本节中，将深入学习这个内容丰富的模块，会看到实现的所有组成部分，还会看到它们如何与模型的其他模块及数据结构进行交互。在实现中，会考虑把 AccountService 作为详细阐述的候选模块。我们现在对这个模块的大部分实现都已

经很熟悉了。[1] 尽管如此，还要往这个实现里引入新类型，使其更加领域友好，同时也集成了前面几章中分别介绍的众多概念。这是非常细节的练习，在接下来的几个章节将逐步探索这个模块。

发布的代数

我们已经看到了代数 API 设计。[2] 我们定义的所有模块都拥有代数，这些代数将作为模块用户绑定的上下文。清单 5.1 详细描述了 `AccountService` 的代数。让我们从上下文的角度来看看模块是怎么构成的。

- *名称*：`AccountService` 是一个名称，它也会体现在领域词汇表中。我们将其作为模块的名称来使用。
- *操作名*：使用 `open`、`close`、`debit`、`credit` 等作为操作名。它们同样直接来源于领域的通用语言。
- *trait 作为容器*：使用 trati 作为模块定义的容器。在 Scala 中，trait 允许定义操作（包括可选实现）以及与其他模块的混合组合。
- *类型别名*：任何发布在模块里的内容都应该有领域友好的名字。类型也不例外。可以定义领域友好的类型别名来隐藏复杂的实现。
- *参数化*：只要可能，用参数化的类型来定义模块。可以在实现模块时提供自己固定的类型。
- *组合性*：模块的所有操作都将返回类型 `AccountOperation`，它是一个赋值的抽象而不是一个值。[3] 这有助于函数具备更好的组合性。看下 `transfer` 方法的定义就会明白，它利用类型的组合从其他操作的代数定义了一个新的操作 *only*。
- *独立实现*：整个模块的定义不包含任何实现细节。从第 3 章开始就知道为什么在定义抽象时需要把代数和实现拆分开。
- *明确的合作*：如果需要在模块内与其他模式进行合作，那就明确说明，这样用户才能清楚模块的整体范围。在 `AccountService` 中，明确表明自己需要 `AccountRepository`（第 3 章中谈到的 Repository 模式）的服务来提供账户管理功能。而且类型定义很明确地表示我们将通过 `Reader` monad 使用依赖注入，使得 `AccountRepository` 的服务在模块中可用。[4]

---

1 我们从第3章就开始讨论这个领域服务了。

2 我们在第3章中讨论了代数API设计。

3 第3章中我们已经讨论过赋值的抽象与值之间的不同。

4 在这里我们将使用Kleisli，而不是第3章中所讨论的`Reader`抽象。它们是相等的；这同时也是个很好的练习，可以尝试证明`Reader` monad可以用Kleisli的方式实现。Kleisli在第4章中有被讨论到。

清单 5.1 模块 AccountService 的代数

NonEmptyList 确保在分离的左偏数据构造器上不会取到空列表。之所以在这里使用一个列表而不是字符串，是因为服务的底层实现可能会由于各种失败而产生一系列的错误。正如在第 4 章中所讨论的，这里可以用 applicative 模式来做作用处理。本章的代码库中包括相关的案例。本节后面的贴士还描述了选择 Scalaz 中右偏的 Either 作为返回类型。

模块定义——领域友好的名称以及类型参数化

```
trait AccountService[Account, Amount, Balance] {
  type Valid[A] = NonEmptyList[String] \/ A
  type AccountOperation[A] = Kleisli[Valid, AccountRepository, A]

  def open(no: String, name: String, rate: Option[BigDecimal],
    openingDate: Option[Date], accountType: AccountType):
      AccountOperation[Account]

  def close(no: String, closeDate: Option[Date]): AccountOperation[Account]
  def debit(no: String, amount: Amount): AccountOperation[Account]
  def credit(no: String, amount: Amount): AccountOperation[Account]
  def balance(no: String): AccountOperation[Balance]

  def transfer(from: String, to: String, amount: Amount):
    AccountOperation[(Account, Account)] = for {
    a <- debit(from, amount)
    b <- credit(to, amount)
  } yield ((a, b))
}
```

领域友好命名的操作定义

组合性——transfer 被定义为其他操作的组合。因为将返回类型作为方法计算，所以这也就具备了可行性。

图 5.3 描述了 AccountService 中一个模块的不同部分。它省略了操作，但将定义自己的模块时需要记住的事项做了重点描述。

模块实现

模块的实现是其代数的解释程序。我们知道对于代数可以有多种实现。应用的不同部分可以使用其中任何一个实现。作为一个设计师，知道在软件工程里有一个优秀实践，就是尽可能迟地提交具体实现。

图 5.4 描述了模型中模块代数与实现的依赖。

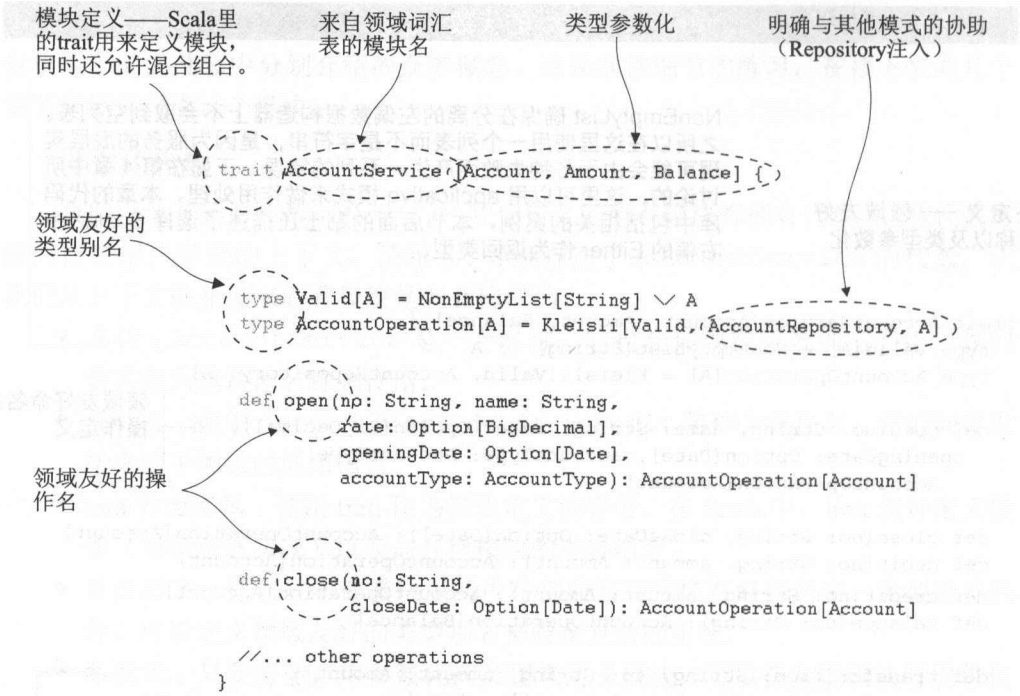


图 5.3 所附的文本中讨论的是模块定义的重要部分。本图展示了定义模块时所需要检查的部分。

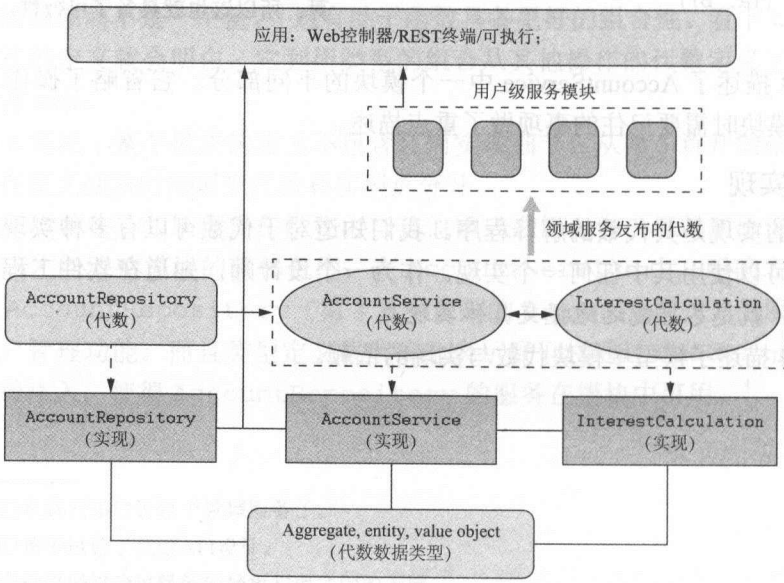


图 5.4 模型中模块的依赖结构。从 A 到 B 的箭头表示 B 使用 A。在模型中，唯一使用模块实现的部分是终端用户应用。模型的其他部分仅基于模块代数开展工作。

这里的 *Application* 方框是终端用户的工具。注意它的依赖。*Application* 是唯一对服务实现[1]有依赖的上下文。模型中其他所有抽象依赖的都是代数。它们都不会将实现泄露给其他模块。同时这也是需要知道的模块化设计的最重要的部分。如果说本节有什么内容是必须掌握的，那就是必须非常小心，不要泄露模块的实现细节。如果实现泄露出去，就意味着模块将可能彼此耦合。这将阻碍模块实现的独立进化。

图 5.4 的依赖结构中，AccountService 依赖于 AccountRepository，但是这个依赖仅停留在代数水平。在生产系统中，可能有一个基于企业级数据库的 AccountRepository 实现。但在测试时，可以将其置换出来，并注入某个轻量级的在内存中的实现。之所以可以这样做，就是因为模块 AccountService 完全与 AccountRepository 的实现解耦。本书的所有模块的完整实现都可以在代码库中查阅到。清单 5.2 展示了 AccountService 解释程序中一些方法的实现样例。

清单 5.2 AccountService 解释程序

```
package domain
package service
package interpreter                                    ← 实现是封装在解释程序中的

import java.util.{ Date, Calendar }

import scalaz._
import Scalaz._
import \/._
import Kleisli._

import model.{ Account, Balance }                      ← 引入模型元素与存储代数
import model.common._
import repository.AccountRepository

class AccountServiceInterpreter extends
  AccountService[Account, Amount, Balance] {

  def open(no: String,                                 ← 用 Kleisli 来注入存储
    name: String,
    rate: Option[BigDecimal],
    openingDate: Option[Date],
    accountType: AccountType) = kleisli { (repo: AccountRepository) =>
```

1 更多内容参见5.2.2节。

```
    repo.query(no) match {
      case \/-(Some(a)) =>
        NonEmptyList(s"Already existing account with no $no").left[Account]
      case \/-(None)    => accountType match {
        case Checking =>
          Account.checkingAccount(no, name, openingDate, None,
            Balance()).flatMap(repo.store)
        case Savings  => rate map { r =>
          Account.savingsAccount(no, name, r, openingDate, None,
            Balance()).flatMap(repo.store)
        } getOrElse {
          NonEmptyList(s"Rate needs to be given for savings
      account").left[Account]
        }
      }
      case a @ -\/(_) => a
    }
  }

  def close(no: String, closeDate: Option[Date]) =
    kleisli { (repo: AccountRepository) =>
    repo.query(no) match {
      case \/-(None) => NonEmptyList(s"Account $no does not
    exist").left[Account]
      case \/-(Some(a)) =>
        val cd = closeDate.getOrElse(today)
        Account.close(a, cd).flatMap(repo.store)
      case a @ -\/(_) => a
    }
  }

  //.. other operations
}

object AccountService extends AccountServiceInterpreter
```

分离右侧的模式匹配——在Scalaz中右边是\/-

用于支票与储蓄账户创建的智能构造器

返回一个NonEmptyList的分离左侧

我们可能会发现，实现中有些事项与第 3、4 章中所开发的版本有所区别。[1]

- *Kleisli*：用 `Kleisli` 来做 `AccountRepository` 的注入，而不是 `Reader` monad。从语义上来看两者是一样的；可以用 `Kleisli` 的方式实现 `Reader`（这也正是 Scalaz 所做的）。[2] 而使用 `Kleisli` 的好处是可以得到对我们有帮助的组合器（也是在第 4 章中所学的）。
- *右偏的 Either*：服务方法的返回类型是来自 Scalaz 的右偏[3] `Either` 类型。它被命名为 `\/` 的同时提供了便利的中缀语法。`\/` 是一个 monad，它提供了便

1 这些区别就是我们逐步所做的提升。

2 Scalaz：https://github.com/scalaz/scalaz 。

3 右偏意味着所有的组合器诸如`map`都将工作在类型的右侧投影。这在用`\/`来做校验时会非常有帮助。我们假设右侧持有经过验证的值，同时高阶函数能被映射到它的上面。

利的组合器来转换到 Scala 标准类，诸如 Either 和 Option。到目前为止，我们已经看到了许多用来做校验的抽象（比如，在失败是一个合法选项的场景中使用案例——Try、Either 和 scalaz.\/）。每一个都有其各自的利弊，在下面的贴士中有相应的汇总。

- 智能构造器：注意用于创建支票与储蓄账户的智能构造器的使用。如第 4 章中所述，这能防止 Account 的子类型的实现泄露到服务的实现中。
- 依赖：服务的实现依赖于模型元素（Account、Balance，等等）以及 AccountRepository 的代数(不是实现)。模型元素的完整实现请查阅代码库。

scala.util.Try 与变体——应该用哪个

在用 Scala 处理类似校验这种情况时，会面临很多种选择。有些来自 Scala 标准库，有些来自 Scalaz。让我们来总结下它们的使用（优缺点），这样作为设计者就知道如何为模型做明智的选择。

- scala.tuil.Try：Try 是专门用于那些可以抛出异常的 API。Try 的 Failure 数据结构使用 Throwable 作为参数。这看上去跟 FP 的核心概念有点背道而驰，FP 并不鼓励把异常从函数中泄露出去。但我们会发现 Try 在处理那些会抛出异常的扩展库（特别是 Java 库）时就非常有用。而采用 Try 的另外一个重要方面是因为它本身是标准库的组成部分。本书中反复说明 Try 是一个 monad，因为它有一个 flatMap。不过从严格意义上来说，Try 违反了 functor 构成的一条法则，在 SI-6284 中也提到了这一点（https://issues.scala-lang.org/browse/SI-6284）。这看起来只是理论上的探讨，也不会影响到在大部分场景中的使用。不管如何，Try 在函数式编程的世界里将异常作为作用来处理，同时会牺牲一些 functor 构成的基本法则，知道这些对我们总是有帮助的。
- scala.util.Either：可以用 Either[Throwable,A] 代替 Try[A]，它会发挥同样的作用。但如果使用 Try,就可以得到一些封装好的组合器，而用 Either 的话，可能要自己写代码。Either 也不是 monad，尽管可以通过 LeftProjection 或 RightProjection 将其当作 monad 来使用，但总是感觉不够灵活。
- scalaz.\/：这是 scala.util.Either 的右偏变体。它是 Scalaz 的一个部分，同时也是一个 monad。它包含了组合器，同时提供了与 scala.util.Either 的无缝对接。因为实现了具有单子作用的校验逻辑，所以它是一个非常有用的抽象（参见第 4 章中关于单子作用与 applicative 作用的区别）。

- scalaz.Validation：顾名思义，这是处理验证最常用也是最给力的技术。Validation[E,A] 是一个 applicative（见第 4 章），它通过一个 Semigroup 实现，内建了对错误收集的支持。如果需要收集所有错误并通过 API 一次性返回，那就使用这个抽象。如果需要单子序列，那就使用 scalaz.\/。

5.2.2 模块的构成

就像之前所讨论的，Scala 里的模块通过基于混合的方式构成。这也是我们所说的 Scala 为模块提供头等支持的原因之一。以下是与个人银行业务相关的小型子集案例：

```
trait InterestCalculation[Account, Amount] {
  def computeInterest: Kleisli[Valid, Account, Amount]
}
trait TaxCalculation[Amount] {
  def computeTax: Kleisli[Valid, Amount, Amount]
}
```

InterestCalculation 和 TaxCalculation 这两个不同的模块提供了对客户账户中余额分别计算利息和所得税的运算规则。但通常情况下，需要将它们组合到一起进行计算（因为需要遵守特定国家的计算规则）。可以定义一个更大的模块来组合 InterestCalculation 和 InterestCalculation：

```
trait InterestPostingService[Account, Amount]
  extends InterestCalculation[Account, Amount]
  with TaxCalculation[Amount]
```

然后为这个合成的代数提供一个组合解释程序：

```
class InterestPostingServiceInterpreter extends
  InterestPostingService[Account, Amount] {
  def computeInterest = //.. implementation

  def computeTax = //.. implementation
}

object InterestPostingService extends InterestPostingServiceInterpreter
```

在代码库中可以找到完整实现。

5.2.3 模块的物理组织

现在已经知道用模块的方式来组织模型的优点，也看到了将模块的实现和代数进行解耦的好处，下面将聚焦在如何组织包结构以便能正确地模块化。良好的包装会带来良好的抽象可视化，这也是编写有良好弹性的代码的一个重要方面。

本章给出的策略只是如何将模型组织到模块中的一个例子。根据模型不同的复杂性，可以随意做些变化。核心思想是要了解良好模块化的代码给模型所带来的好处。在讨论结构前，先看下图 5.5，它描绘了一个单一模块的整体结构，展示了如何组织模块的代数、实现、合作模块以及最终的终端用户应用。

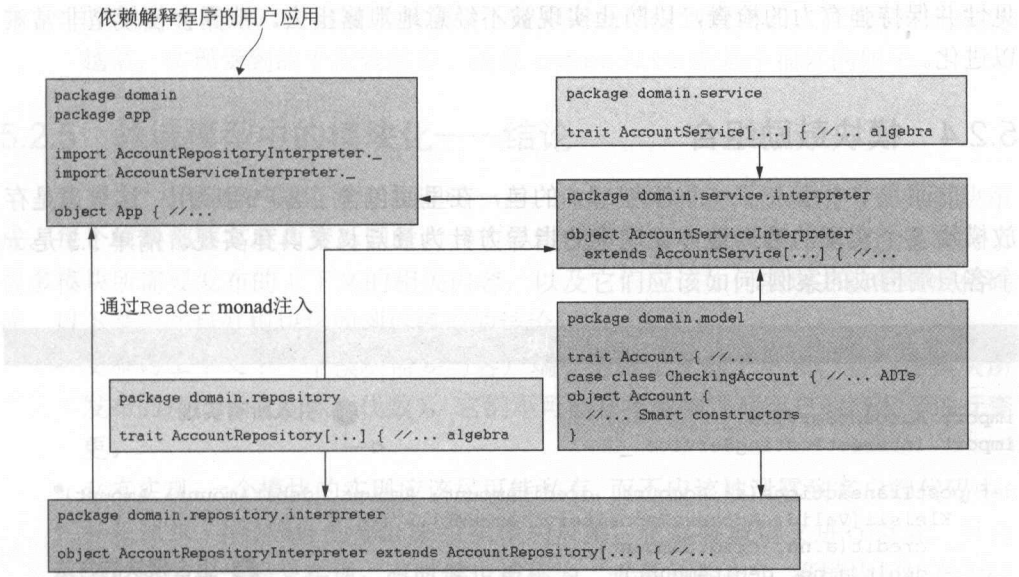


图 5.5 与合作者一起组织模块。每个箭头都描述了模块间的依赖。这张图展示了包 model 里的领域元素、包 service 里的服务模块代数以及包 interpreter 里的解释程序。解释程序仅被用于终端用户应用，该应用被表示为 App 并被包含在包 app 中。

我们有基础包 domain，在它下面还有一系列的子包，它们以模块的方式各自包含了特定的工具。

- *模型元素*：包被命名为 model，包含了建模为实体、聚合以及值对象的代数数据类型。它还包含了伴生对象，为核心模型元素提供智能构造器、透镜以及其他方面支持的抽象。
- *仓储*：仓储被保留为单独的模块，我们将这个包命名为 repository。由于一个仓储会产生一个模块，也会有相对应的实现，因此代数将保留在 repository 内部，而解释程序将被安置到子包 interpreter 或 implementation 中。

- 领域服务：也就是仓储的组织。用一个包 service 来对应所有代数，用 interpreter 来对应所有实现。
- 应用：这个命名为 app 的包是用于终端用户的应用。在当前的上下文中，它只是个占位符。在现实生活中，它将拥有终端用户应用托管的包结构。如果使用像 Play 这样的框架，将拥有一个符合 Play 应用结构的包结构。

这只是一个完整的领域驱动设计的包结构的案例。如果模型的复杂度不同，就会需要做一些变化。我们把所有服务模块保持在一个级别。如果在同一个边界上下文中有大量的服务，可能就需要借助针对功能的子包。如果在同样的边界上下文中有不同的功能模块，另一种方式是在顶层就拆分成不同的子包。但重要的是控制可见性并保持强有力的检查，以防止实现被不经意地泄露出去。泄露会使模型非常难以进化。

5.2.4 模块鼓励组合

前面章节中引入了一个名为 app 的包，在里面包含了客户端应用，这里也是存放模块各个实现的唯一地方。这里的指导方针为*最后提交具体实现*。清单 5.3 是一个客户端构成的案例。

清单 5.3 在客户端应用中使用模块

```
import AccountService._
import InterestPostingService._

def postTransactions(a: Account, creditAmount: Amount, debitAmount: Amount)
     Kleisli[Valid, AccountRepository, Amount] = for {
  _ <- credit(a.no, creditAmount)
  d <- debit(a.no, debitAmount)
} yield d

def composite(no: String, name: String, creditAmount: Amount, debitAmount:
     Amount): Kleisli[Valid, AccountRepository, Amount] = (for {
  a <- open(no, name, BigDecimal(0.4).some, None, Savings)
  t <- postTransactions(a, creditAmount, debitAmount)
} yield t) andThen computeInterest andThen computeTax

val x = composite("a-123", "John k", 10000, 2000)
  (AccountRepositoryInMemory)
```

❶ 引入所有实现

❷ 组合模块的方法

模块间的组合

AccountRepository 实现（具体实现参见代码库）

下面列出了该实现的一些结论。

- 提交模块实现：这是需要使用模块具体实现的地方。在代码 ❶ 中引入的两个实现就是做这个的。同样当调用 composite 命令 ❷ 时，也给它传入了 Ac-

countRepository 的实现。

- *组合模块的方法*：在 postTransactions 函数中，组合了 AccountService 的方法。之所以可以这样做是因为每个方法都返回了一个 monad。这是通过方法返回作用所取得的值，而不是返回一个具体值。
- *模块间的组合*：建立更大程度的服务或抽象时，经常会在实现中用到多个模块。建立可组合抽象的窍门是始终关注模块方法返回的类型。在 composite 函数中，对 AccountService 以及 InterestPostingService 中的方法进行组合。composite 里的 for 表达式返回了一个 Kleisli，它可以很自然地被链接到模块 InterestPostingService 中的方法，因为后者返回的也是 Kleisli。这就是从类型匹配中所能享受到的快乐。在类型中封装的越多，实现受到的干扰就越少。函数 composite 就是个很好的例子。

5.2.5 领域模型中的模块化——结论

章节 5.2 中所有的讨论内容都说明一点，良好架构的软件需要由一系列模块组成。每个模块都要高度内聚，同时与其他模块及环境最小地耦合。我们已经学习了很多模块所需要发布的上下文的相关内容，以及它们应该如何实现并被组织成代码库。以下是一些良好模块化的领域模型的结论。

- *发布的上下文*：一个模块需要与客户端有清晰定义的上下文。它们是模块所发布的接口（也被称为代数），它们不可能在不影响现存客户的情况下进行变更。
- *私有实现*：一个模块的实现应该尽可能私有，而不应该被泄露到客户端代码中。
- *模块组织*：用 Scala 的 trait 来组织模块规则，依靠领域语义进行组合。只在客户应用水平提交实现，有时候也被称为“世界的尽头”。将模块保持在不同的包中，在包水平上将模块代数与实现隔离开。
- *组合性*：客户要能够将模块组合到一起，并通过小型模块演化出更大型的模块。

5.3 类型类模式——模块化的多态行为

在设计领域模型时，经常需要实现涉及很多对象的具体行为。这些对象之间可能没有任何联系，但它们需要共享这个通用行为。一个常见的例子就是串行。许多领域对象，比如 Account、Customer 以及 Bank，需要在多个上下文间传递它们时，必须成为串行的。5.4.2 节讨论多个边界上下文间的通信时，就会看到一个这样的案例。

使用对面相对范式实现这个的一个方式是借助子类型多态性。对基本行为使用trait，然后每个类可以为特定行为提供一个实现。Java 中的 java.io.Serializable，会在某个点用来提供特定类的序列化行为。这里有一个 Scala 里实现方式的案例：

```
trait Serializable[T]
trait Account extends Serializable[Account] { //..
case class Customer(..) extends Serializable[Customer] { //..
```

这种方式有几个缺点。第一、它将核心领域抽象与它所需实现的非核心行为耦合在一起。第二、需要在类定义的地方详细说明这个行为。必须访问 Account 的源代码，才能使其序列化。有一些其他模式可以克服这个缺点，但它们都及其繁琐而且难以实现和维护。

类型类为解决这个问题提供了另外一种方式。基本思路是通过多态行为使类彼此之间没有任何关系。有些时候这也被称为自组织多态性，它通过参数化多态的方式来实现。

让我们看一个个人银行领域的小例子。当我们生成报告时，需要按特定格式显示领域元素。对于某些金融实体，比如证券或货币，它们有 ISIN 代码，这就需要一个特定格式，而且客户账户与金融机构的显示格式是不同的。最终，需要设计一个协议来显示领域模型中的所有元素（行为与对象），我们将这个协议命名为 Show。Show 定义了一个方法 shows，它在每个需要定制显示协议的领域抽象中都会被实现。

```
trait Show[T] {
  def shows(t: T): Try[String]
}
```

如何在不改变定义的情况下，在多个不相干的抽象中实现这个行为呢？这就是类型类模式所具备的能力。类型类模式首先在 Haskell 中得到实现，Scala 也提供了一个途径来编码这个模式。让我们来看一下如何对 Account 类做这个编码。同样的技巧也可以被沿用到其他类上：

```
case class Account(no: String, name: String, dateOfOpening: Date = today,
  dateOfClosing: Option[Date] = None,
  balance: Balance = Balance(0))
case class Customer(no: String, name: String, address: Address,
  email: String)
trait ShowProtocol {
  implicit val showAccount: Show[Account]
```

```
  implicit val showCustomer: Show[Customer]
  //..
}
```

我们已经有了类 Account，并且定义了一个模块，它为领域元素的 Show 协议定义了代数。下一步是为每个协议元素提供实现：

```
trait DomainShowProtocol extends ShowProtocol {
  implicit val showAccount: Show[Account] = new Show[Account] {
    def shows(a: Account) = Success(a.toString)
  }
  implicit val showCustomer: Show[Customer] = new Show[Customer] {
    def shows(c: Customer) = Success(c.toString)
  }
  //..
}
object DomainShowProtocol extends DomainShowProtocol
```

现在我们已经拥有了一个模块，DomainShowProtocol，它实现了在协议 ShowProtocol 中定义的代数。大家很快就会明白为什么我们已经隐式地制定了实现。

我们已经在单独的模块中定义了协议实现。元素和协议被彻底地解耦，这也就解决了 OO 实现的问题。接下来看一下如何使用类型类模式：

```
object Reporting {
  def report[T: Show](as: Seq[T]) = as.map(implicitly[Show[T]].shows(_))
}
```

假设报告模块中包含一个 report 函数。report 将显示通过特定协议传输给它的元素集合。我们可以传入 Show 协议的具体实现，同时预期 report 将显示相应的元素。在这个例子中，使用 Scala 上下文绑定语法使 Show 协议成为一个隐式参数。当调用 report 函数时，如果在可用范围内，编译器会自动传递为协议传递一个恰当的隐式的值。[1] 下面的例子为 Show 协议引入一个实现模块，当调用 report 时它会被自动传入。我们为 Show 类型类使用协议编码，这使得 API 更加简洁而且类型安全。而当调用 report 时，如果编译器不能在范围内找到任何匹配的隐式值，它就会报错。

1 范围是指协议可见的范围。在Scala中它是用隐式参数的方式实现的。访问www.scala-lang.org/files/archive/spec/2.11/07-implicit- parameters-and-views.html并阅读“隐式参数”，了解Scala中隐式范围解析的更多细节。

```
import DomainShowProtocol._

val as = Seq(
  Account1("a-1", "name-1"),
  Account1("a-2", "name-2"),
  Account1("a-3", "name-3"),
  Account1("a-4", "name-4")
)

Reporting.report(as)
```

❶ 为 Show 引入所有领域协议的默认实现

❷ 使用默认协议实现

总的来说，在领域建模中使用类型类的好处有哪些？这里列出了 3 个主要方面：

- 模块化：类型类将相关行为分组到模块内，可以根据领域元素的实现进行导出。这使得代码可以根据相关行为分组进行模块化，而不是根据相关对象。这个理念非常适合函数式编程的原则。
- 自组织多态性：使用类型类可以为无关抽象中所选择的行为实现多态。在前面的例子中，实现了一个协议，它为领域元素如 Account 和 Customer 创建定制的显示，两者之间是完全不相干的。最厉害的是可以在现成的抽象上实现此类行为，而这是子类型多态所做不到的。
- 协议选择：在前面的例子中，引入了所有协议的默认实现 ❶，在 report 的调用 ❷ 中它会被隐式使用。根据 Scala 的隐式解析规则，在调用 report 方法时还要提供协议相应的实现。这使我们在协议实现上具备了类型类的编码层，并插入最适合上下文的那个实现。

5.4 边界上下文的聚合模块

在第 1 章中已经讨论过领域驱动设计的边界上下文以及其不同的模式。接下来让我们更深入地探讨在领域模型模块化中边界上下文所扮演的角色。在对复杂领域模型模块化的过程中，边界上下文可能是最重要的概念了。任何非传统的领域模型都不仅仅只是一个模型：它会由多个模型组成。哪怕是个人银行领域，也包含了好几个小型模型，比如账户管理、报告服务以及后台管理。这些都是完全不同的功能，它们又有完全不同的实体、行为以及术语表，它们的实现很可能会使用完全不同的领域与数据模型，哪怕是这些模型中命名相同的事物都可能有完全不同的含义以及生命周期。

回顾一下 Account 抽象。当我们想到银行业务核心模型中的 Account 时，就知道它是一个实体（在前面章节中已经出现多次）。一个账户的生命周期是需要管理的，包括对应到账户的编号。在一个报告上下文中，并没有将账户当实体来管理。在一个报告模块中 Account 是一个值对象，它的值需要能够被打印出来，同时通过结构化的数据模型进行管理以便于更快地查询和生成报告。每个应用都需要有一个认证模块，而在认证的上下文中 Account 又有完全不同的含义：它是一个必须对系统使用进行认证的用户账户，与银行所维护的用户账户完全不同。

模型中的每个小模型各自形成了一个个边界上下文，每个边界上下文都有以下特点：

- 有独立的相干数据以及领域模型。
- 可能会有与其他边界上下文中的元素相似的名字，但语义完全不同。
- 拥有只在特定上下文中有效的通用语言。
- 与其他边界上下文尽可能小地耦合，且需要被明确地定义。

我们刚刚讨论过模型中模块。为什么不把功能当作独立的模块呢？模型在一个单独的边界上下文中需要保持高度一致。我们不可能在一个边界上下文中使两个元素有相同的名字却有不同的语义。这在数据模型上会产生歧义，对于边界上下文来说也是非常滑稽的一件事。

注意：更多关于边界上下文如何与领域驱动设计相联系的细节，可以参考 Eric Evans 所著的 *Domain-Driven Design: Tackling Complexity in the Heart of Software*（Addison-Wesley Professional, 2003 年）和 Vaughn Vernon 所著的 *Implementing Domain-Driven Design*（Addison-Wesley Professional, 2013 年）。

5.4.1 模块与边界上下文

模块与边界上下文是怎样一种关系呢？边界上下文是粒度更高级别的一种体现，它通常包含多个模块。图 5.6 很清晰地用实例解释了这种关系。这里还有一些边界上下文之间通信的注释，马上我们就会讨论到。

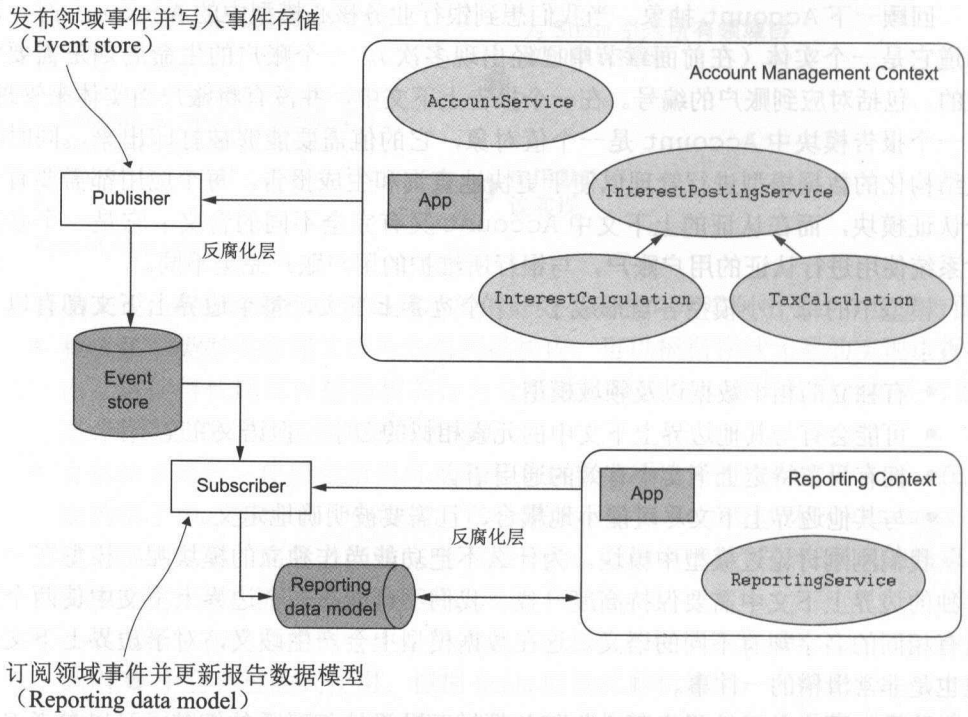


图 5.6 一个边界上下文包含多个模块。Account Management 与 Reporting 是两个包含多个模块的上下文。Publisher 与 Subscriber 两个矩形描述了背景图，并且建立了两个边界上下文的通信协议。两个上下文之间并没有直接的联系。所有通信都要通过背景图中明确发布的协议才能进行。

5.4.2 边界上下文间的通信

在 *Domain-Driven Design* 一书中，作者 Eric Evans 提到一些方法，用于在两个边界上下文之间建立通信路径。其中一个是通过反腐化层（anticorruption layer），它在两个边界上下文之间建立了一个明确的通信层，可以防止边界上下文被其他上下文的实现直接干扰，并在两个组件之间提供隔离。除了反腐化层，Evans 还在书中提供了其他一些技术，如果有兴趣也可以进一步了解。但基本的思路都是防止在边界上下文之间出现不可控的通信。

在两个边界上下文之间进行通信时，抽象的语义就会打破，就像之前在账户管理与报告服务上下文中 `Account` 的例子。在这两个上下文中，名字是相同的，都是来自领域词汇表。但是属性变了，生命周期变了，身份管理也变了。很明显，在账户管理里的 `Account` 抽象与报告服务中的不同。但它们是有关联的，当需要报告服务打印在账户管理上下文中创建的账户详情时，需要建立两者之间的映射。

消息是用于映射且不会在这两个上下文间产生耦合的最有效的技术手段。消息为类型提供了一个天然的映射空间，在这里我们解构抽象并将一个抽象映射到另一个。消息的另一个好处是异步处理，这使我们可以在空间和时间上进行解耦。这两个上下文在时间和空间上都被解耦，同时异步消息在两者之间扮演完美的通信黏合剂。唯一的要求就是需要一个协议，能被两个上下文都理解，同时能响应它们所接收到的消息。[1]

在这个例子中（见图 5.6），使用发布—订阅模型作为消息协议来设计反腐化层。账户管理上下文执行命令，发布事件并存储到事件日志。报告上下文是这些事件的订阅者，并且在收到这些事件之后更新它的存储。这里有两个边界上下文，它们都在其数据与领域模型的约束内进行操作，并通过一个明确发布的通信层小心地保护着自身的不变性。第 6 章中会讲述这个使用场景的案例。

5.5 模块化的另一个模式——free monad

模块化的主要目的是确保我们在修改模型的一个部分时不会影响（或使影响最小化）其他无关部分。我们通过合适的抽象来保证这一点。一个模块就是这样一个抽象，它还会扮演其他抽象的容器。前面章节中的 AccountService 就是这样的例子。我们快速回顾一下模块的特点：

- 具有明确指定代数的类型与函数的集合。
- 对语义相关的领域行为进行建模。
- 可以有多个与代数解耦的解释程序或实现。

在 5.2 节中已经看到如何对领域模型进行模块化的例子，而且学习了账户管理模块 AccountService，还看到了它发布的代数以及实现的案例。

AccountService 用到了另外一个模块 AccountRepository，它提供了 Account 聚合函数来与存储进行交互。本节将聚焦在该存储上，同时讨论一个模块架构与实现分离的高级模式。接下来首先会看到之前所使用的标准接口 / 实现分离模式。然后将试着用函数式编程里的高级技术在另外一个水平上重新实现。

注意：这是个高级的专题，所以如果是首次阅读本章，也可以跳过这一节。

5.5.1 账户存储

让我们从该模块的规则内容入手，其中一个被发布到客户端并被 AccountService 所使用。

1 感谢Martin Thompson在Twitter上分享这个思路。

清单 5.4 AccountRepository 模块定义

使用领域元素 Account 和 Balance。

NonEmptyList 是一个列表抽象，其限制是不能为空。我们通常会考虑使用这种隐式编码一些约束的抽象。

模块定义——每个方法返回 Scalaz 的右偏 Either。

```
import model.{ Account, Balance }

trait AccountRepository {
  def query(no: String): \/[NonEmptyList[String], Option[Account]]
  def store(a: Account): \/[NonEmptyList[String], Account]
  def balance(no: String): \/[NonEmptyList[String], Balance] =
    query(no) match {
      case \/-(Some(a)) => a.balance.right
      case \/-(None) => NonEmptyList(s"No account exists with no
     $no").left[Balance]
      case a @ -\/(_) => a
    }
  def query(openedOn: Date): \/[NonEmptyList[String], Seq[Account]]
}
```

一个模块定义可以有多种实现，让我们从其中最常见的一个开始，它实现了之前的 trait，并通过使用内存中的可变 Map 为每个方法定义提供相关语义。

清单 5.5 AccountRepository 一个内存中的实现

```
import scala.collection.mutable.{ Map => MMap }
import model.{ Account, Balance }

trait AccountRepositoryInMemory extends AccountRepository {
  lazy val repo = MMap.empty[String, Account]

  def query(no: String): \/[NonEmptyList[String], Option[Account]] =
    repo.get(no).right
  def store(a: Account): \/[NonEmptyList[String], Account] = {
    val r = repo += ((a.no, a))
    a.right
  }
  def query(openedOn: Date): \/[NonEmptyList[String], Seq[Account]] =
    repo.values.filter(_.dateOfOpen == openedOn).toSeq.right
}

object AccountRepository extends AccountRepositoryInMemory
```

现在有了模块的接口与实现，可以运行下面的代码片段并得到一些结果：

```
import AccountRepository._
import scalaz.syntax.std.option._
val account = checkingAccount("a-123", "John K.", today.some,
  None, Balance(0)).toOption.get
val dsl = for {
```

创建支票账户的智能构造器

```
    b <- updateBalance(account, 10000)
    c <- store(b)
    d <- balance(c.no)
  } yield d
```

Account 模块中的 updateBalance 函数，它将更新账户的余额并返回 NonEmptyList[String] \/ Account。

现在有了模块 AccountRepository 的实现以及每个方法定义的语义。模块的每个方法均返回一个 monad 的分离（scalaz.\/）。因此我们自然可以建立自己的小型 DSL，用 for 表达式创建账户、存储它们、查询并获取余额，等等。

运行之前的代码所得到的结果是 \/-(Balance(10000))。因为 \/ 是一个 monad，这个序列将通过 flatMap 和 map 来进行串行计算并最终给出结果。

5.5.2 使它免费

5.2.1 节中的实现解决了模型中接口与模块实现解耦的大部分问题。本节将提供一个新的方式来解耦模块功能：将领域行为建模为纯数据并提供解释程序，它将在特定上下文中对这些数据开展工作。这与编译器的工作方式很类似；将程序转变成一个抽象语法树（AST），它是一个纯数据，然后定义独立的解释程序来操作树并执行各自的转化，比如最优化、编译到字节码以及打印。

在为特定的领域行为集设计模块时，可以考虑将每个行为都建模为纯数据，而不考虑操作上它将如何被执行。一旦定义出所有模块的行为，就将它们作为闭合的代数数据类型打包到一起。通过 ADT，现在可以将元素组合成复杂的递归值，并对更复杂的领域行为进行建模。这都是关于计算结构的创建，我们并没有看到任何关于数据类型如何被执行的实现。执行是以解释程序的方式进行，而且在这个模块化的模式中它是一个完全独立的阶段。在拥有数据类型之后，可以定义多个解释程序，每一个都有其自身的执行语义。可以直接执行它们，也可以通过应用领域规则来执行一些优化，或者也可以把所有数据打印出来用于审计跟踪。

我们将这个称为 *Free Monad* 模式。在后面的章节中会看到实现的例子。作为到目前为止所学内容的总结，下面列出了基于 free monad 计算结构的主要特性：

- 行为体现为纯数据并组成闭合的代数数据类型。
- 计算的创建与执行之间的严格隔离。
- 计算的执行以解释程序的方式体现，对同一个结构可以有多个解释程序。

我们已经知道 free monad 是如何帮助我们对模型进行模块化的，现在让我们开始下一个步骤，并找出如何在实践中应用这个模式。假设我们已经掌握了 free monad 的工作机制，接下来将学到如何隔离纯数据与模块解释程序。这里故意留下了一些模糊的步骤，不够后面我们还会回来。

- 积木：积木从一系列的行为开始入手，对组成模块代数的独立操作进行建模。它们可以被组合成模块的更大的行为，甚至一个小型的 DSL。在 AccountRepository 的上下文中，可以将独立的操作想象成 DSL 的积木。这跟之前 AccountRepository 的实现中所采用的思路是一致的。区别在于如何对这些行为进行建模，它们不再被建模为模块中的独立函数。用代数数据类型将它们设计为纯数据，马上就会看到相关例子。
- 魔术带：将这些积木串行组合的一种方式是将它们包含在一个单子上下文中。但我们现在还没有 monad，只有一些 ADT 作为纯数据。于是魔术带出现了：我们做了些神奇的事，然后得到一个免费的 *monad*。不用担心其中的细节。在下一节中会看到怎么变这个魔术。现在我们就当作已经有了一个 monad，在里面可以提取每个积木。
- 模块：我们已经有了一个 free monad，现在需要将积木放进 monad 的上下文中。这些提取操作组成了模块的公开 API。因为每个操作都返回一个 monad，所以可以用 for 表达式来将它们组合成 DSL。
- 组合的 *DSL* 返回的是什么呢？在拥有 DSL 之后，我们希望通过 monad 的 flatMap 串行来执行它。一个 monad 的序列会做什么是由 monad 实现的 flatMap 所定义的。凑巧的是 free monad 的 flatMap 不做任何事，它只是把积木当纯数据进行积累并上交积累的结果（也就是 AST）。这也是我们说 free monad 没有语义或外延的原因。[1] 所以现在已经完成第一步，free monad 交出了没有上下文的纯数据。
- 外延：现在已经拥有了 AST，接下来需要提供一个解释程序，使其可以根据需求进行执行。free monad 完成积累部分（也就是 flatMap 所做的），但没有执行，这也是 flatMap 提交 AST 的方式。解释程序将扫描 AST 并定义 DSL 的语义，而且解释程序拥有应用的上下文。如果希望自己的 AST 得到执行，同时返回的结果是错误与值的分离，那就实现这样的解释程序。如果想要异步地执行 AST，那就可以从解释程序中返回一个 Future。解释程序将在 free monad 给纯数据上应用上下文。

前面的步骤在 DSL 结构的定义与解释程序之间引入了一个间接层。在 5.2.1 节的例子中，组成 AccountRepository 操作序列的 DSL 已经拥有了 scalaz.\/ 的单子上下文所定义的执行序列。而在现在的案例中，free monad 并未定义执行序列，而是让解释程序结合 AST 来解决这个问题。

1 所有的语义都被包含在解释程序中。

5.5.3 账户存储——free monad

我们可以去深入学习 free monad 的实现细节。但作为一个用户，我们对其中绝大部分内容都不会太关心。所以让我们从 5.5.2 节中所列出的 4 步中的第一步开始，来将 AccountRepository 实现为一个 free monad。我们会将最后一个步骤(外延)保留到 5.5.4 节中。

定义积木

以下是要在存储上支持的不同操作。我们还是对其进行简化，因为要搞清楚 free monad 如何帮助我们组合 DSL 并将代数与解释程序进行解耦。清单 5.6 定义了所要支持操作的数据类型。

清单 5.6 账户存储 DSL 的积木

```
sealed trait AccountRepoF[+A]
case class Query(no: String) extends AccountRepoF[Account]
case class Store(account: Account) extends AccountRepoF[Unit]
case class Delete(no: String) extends AccountRepoF[Unit]
```

动作的基本 trait

每个动作表示为一个 ADT(纯数据)

在清单中，将存储中的每个动作都定义为一个纯数据元素。每个动作的数据类型各自定义了代数，但没有附带上任何行为方面的语义。[1]

魔术带——获取 free monad

我们现在来讨论一下上一章节中提到的魔术。就像任何魔术一样，公众只需感受其神奇的部分。在本节中，将作为 API 的用户来享受这个魔术。Scalaz 中 API 的实现相当复杂，我们永远不需要为了在领域模型中使用 free monad 而去了解具体细节。唯一需要知道的是如何用 Free 类型从 ADT 中生成一个 free monad：

```
type AccountRepo[A] = Free[AccountRepoF, A]
```

这使 AccountRepo 成为一个 free monad。大家可能会疑惑，为什么把它叫做 *free* monad。根本原因还在策略理论上，我们不打算马上来讨论这事。在 Rúnar Bjarnason 的论文 *Free Monoids and Free Monads* 中对底层概念有非常深入的讲解(http://blog.higherorder.com/blog/2013/08/20/free-monads-and-free-monoids/)，有兴趣的话可以了解一下。

现在已经有了一个 monad，下一步就是将所有操作提取到 monad 的上下文中。其结果是产生一系列的智能构造器，每个都对应一个操作，然后就可以进行组合并

1 这也正是代数数据类型所做的。

建立更高阶的抽象。

定义模块

现在将动作放入 AccountRepo 的上下文中，这将定义模块 AccountRepository，如清单 5.7 所示。与之前的实现不同（参见 5.2.1 节），这个模块并不附加任何解释，比如动作应该如何执行或者它们将返回什么值。每个动作将返回一个 free monad 的计算。这里我们会用到 scalaz.Free.liftF 函数。

清单 5.7 AccountRepository 的模块定义—没有外延

```
import scalaz.Free
trait AccountRepository {
  def store(account: Account): AccountRepo[Unit] =
    Free.liftF(Store(account))

  def query(no: String): AccountRepo[Account] =
    Free.liftF(Query(no))

  def delete(no: String): AccountRepo[Unit] =
    Free.liftF(Delete(no))

  def update(no: String, f: Account => Account): AccountRepo[Unit] = for {
    a <- query(no)
    _ <- store(f(a))
  } yield ()

  def updateBalance(no: String, amount: Amount,
    f: (Account, Amount) => Account): AccountRepo[Unit] = for {
    a <- query(no)
    _ <- store(f(a, amount))
  } yield ()
}
```

liftF 将 Store 操作提取到 AccountRepo 的上下文中

组合 DSL

除了 ADT 提供的基本操作 store、query 和 delete 之外，清单 5.7 中的模块 AccountRepository 还定义了其他的操作，如 update 和 updateBalance。这些都是复杂的领域行为，monad 允许通过基本的 ADT 代数来建立它们。updateBalance 返回一个递归的数据结构，该结构有单独的组件，而这些组件又是通过我们在 ADT 中定义的 Query 和 Store 的基本代数生成的。

借助 monad 组合的力量，现在可以建立围绕 AccountRepository 模块发布 API 的小型 DSL：

```
def open(no: String, name: String, openingDate: Option[Date]) = for {
  _ <- store(Account(no, name, openingDate.get))
  a <- query(no)
} yield a
```

创建账户，存到存储中，提取出来并返回

```
val close: Account => Account = { _.copy(dateOfClosing = Some(today)) }

def close(no: String) = for {
  _ <- update(no, close)
  a <- query(no)
} yield a
```

关闭账户，提取已关闭的账户并返回

这些账户管理函数可以用作领域服务 API 的组成部分。但要真正用起来，还需要为每个操作关联语义。这也是下一节中要讨论的内容。在此之前，让我们先快速地回顾一下需要遵守的相关步骤——用 free monad 编写组合 DSL，使模型的代数与解释具备极大程度的隔离。图 5.7 详细描述了这些步骤。

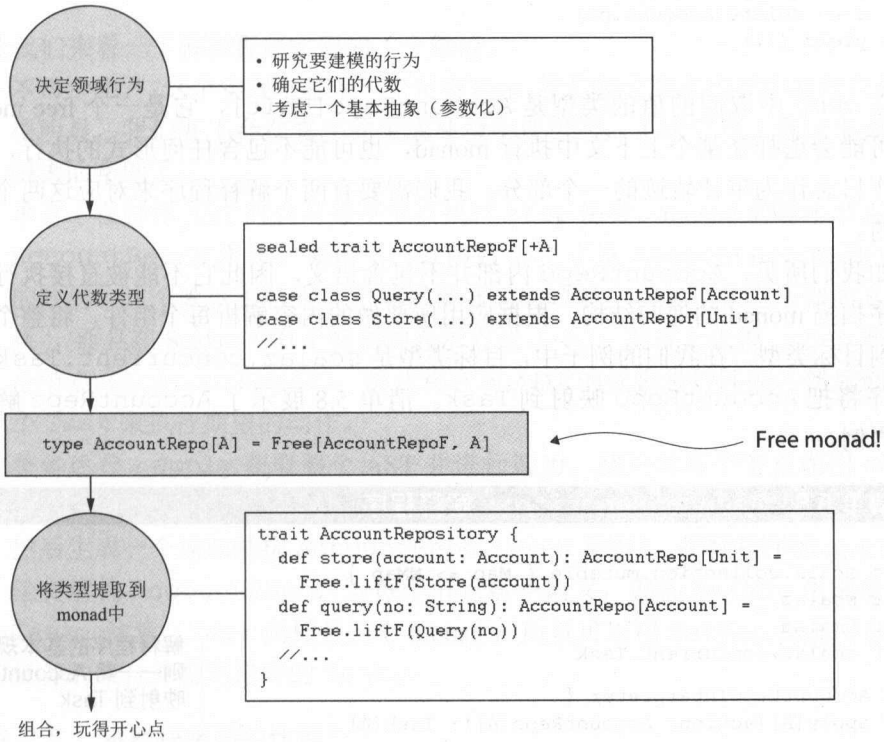


图 5.7 遵守以上步骤并获得基于 free monad 的小型 DSL 实现。每个步骤在相应的文本中有详细说明，这个图可作为一个现成的参考。

5.5.4 free monad 解释程序

现在已经来到了最后一步，我们将执行基于 free monad 的实现。解释程序为每个代数元素提供了外延，它将 free monad 内建在组合的 DSL 里。解释程序可能还会包含副作用。注意，我们已经触碰到了模型的边界，在这里需要与真实世界进行交互。

而真实世界充满了副作用（可能会需要与数据库或文件系统发生交互，它们都内建了副作用）。最好的做法是将所有副作用推到系统边界中，以保持核心代数的纯洁与函数化。

解释程序是应用的上下文组成部分，它将根据我们的期望操纵 free monad 数据结构。我们可能会选择对相同的数据表现形式执行多种操纵，比如下面针对账户存储用之前 ADT 组合而成的代码：

```
val account = Account("a-123", "John K")
val comp = for {
  a <- store(account.copy(balance = Balance(1000)))
  q <- query(account.no)
  c <- delete(account.no)
} yield ((()))
```

在 comp 中取回的值的类型是 AccountRepo[Unit]，它是一个 free monad。我们可能会选择在某个上下文中执行 monad，也可能不包含任何形式的执行，只记录操作日志作为审计轨迹的一个部分。我们需要有两个解释程序来对应这两个不同的目的。

如我们所见，AccountRepo 内部并不包含语义，因此它不能被直接执行。解释程序扫描 monad 的递归结构，根据应用所要做的内容解析每个组件，将整个计算分解到目标类型。在我们的例子中，目标类型是 scalaz.concurrent.Task[1]，解释程序将把 AccountRepo 映射到 Task。清单 5.8 展示了 AccountRepo 解释程序的样例。

清单 5.8 AccountRepository 解释程序

```
import scala.collection.mutable.{ Map => MMap }
import scalaz._
import Scalaz._
import scalaz.concurrent.Task

trait AccountRepoInterpreter {
  def apply[A](action: AccountRepo[A]): Task[A]
}

case class AccountRepoMutableInterpreter() extends AccountRepoInterpreter {
  val table: MMap[String, Account] = MMap.empty[String, Account]

  val step: AccountRepoF ~> Task = new (AccountRepoF ~> Task) {
    override def apply[A](fa: AccountRepoF[A]): Task[A] = fa match {
      case Query(no) =>
```

解释程序的基本规则——将 AccountRepo 映射到 Task

使用可变 Map 存储账户

step 是 AST 每个节点所执行的函数

1 在Scala中Task与Future比较类似，Task的计算能力更强一点。我们会在第6章中更多关于Task的细节。

```
      table.get(no)
           .map { a => now(a) }
           .getOrElse {
             fail(new RuntimeException(s"Account no $no not found"))
           }
    case Store(account) => now(table += ((account.no, account))).void
    case Delete(no) => now(table -= no).void
  }
}

def apply[A](action: AccountRepo[A]): Task[A] = action.foldMap(step)
}
```

now 是 Task 的组合器，它将值提取进 Task[A]

fail 是执行中用于处理失败的组合器

运行单子上下文中所有 AST 节点

让我们来看一下解释程序实现的 3 个结论。

- 不用于生产：这个实现是基于一个可变 Map，我们将它当作内存中的账户存储。很明显，我们并不会在真正的生成环境中使用它，不过它对于测试来说就有很大的帮助了。
- 单步：在解释 AST 时会对每个节点执行 step 函数。monad 的每个节点都是 AccountRepoF 类型，同时 step 函数定义了从 AccountRepoF 到 Task 的映射。这个映射很特别，它是保留结构的映射，也就是我们所知的自然转换[1]，而且在 Scalaz 中有个特别的表示方式（~>）。直观上来说，step 获取一个节点（AccountRepoF），模式将它与 ADT 的元素进行匹配，然后创建一个 Task 来执行期望的动作。
- 整体运行：apply 获取整个 AST 并进行遍历，然后对每个节点调用 step。当处于 DSL 构建阶段时，通过 flatMap 不断积累独立的 AccountRepoF 节点，然后生成一个递归结构 AccountRepo（Free 类型）。解释程序的 apply 方法将解构 AccountRepo，为每个节点赋予语义，然后 flatMap 生成最终的 Task。注意，Task 同样是一个 monad[2]，所以可以用 flatMap 扫描所有独立任务的集合并得到最终的 Task。

练习 5.1 解释程序中的作用[3]

在我们讨论的 free monad 实现中，清单 5.8 中的解释程序返回一个 Task。Task 是一个 monad，可以基于应用的上下文对其进行解释。除了 Task，可能还希

1 自然转化的概念对我们来说不是特别重要。在Rúnar Bjarnason的论文*Free Monoids and Free Monads*（http://blog.higherorder.com/blog/2013/08/20/free-monads-and-free-monoids/）中解释了它的基本概念以及如何关联到free monad。

2 可以查看Scalaz的源代码并了解Task monad是如何实现的。

3 Jan Vincent Liwanag建议做这个练习，在此表示感谢。

望用其他作用的方式解释 free monad。所以为什么不用希望生产的作用对解释程序进行参数化呢？这里有个例子：

```
trait AccountRepoInterpreter[M[_]] {
  def apply[A](action: AccountRepo[A]): M[A]
}
```

在这个练习中将探索用 State monad 实现解释程序。[1] 我们不再使用一个可变 Map 来存储状态，而是将 Map 抽象为 State monad 内部的实现细节。于是它就成为一个不可变的 Map。

```
object AccountRepoState {
  type AccountMap = Map[String, Account]
  type Err[A] = Error \/ A
  type AccountState[A] = StateT[Err, AccountMap, A]
}
```

注意，该实现需要使用一个 Map 来存储账户细节。用 scalaz.\/ 来处理错误，最后得到一个收集错误的 State monad。

用 AccountState 而不是 Task 实现一个 AccountRepoInterpreter。当传入 free monad 并执行解释程序时，它将返回一个 AccountState。

5.5.5 free monad——重点回顾

就像建模与设计里的其他模式一样，free monad 也是工具箱中很重要的一个部分。如前文所说，在今天的 Scala 社区里它并不是一个很常用的模式。但我们现在应该已经意识到它的作用。社区中的专家们正在大力倡导这种模式，而且不少库已经在使用这个优雅的设计模式。

下面是该模式给模型带来的 3 个主要好处：

- 模块化与可测试性：可以基于 free monad 模块化应用。与其他设计模式相比，它可以用更强壮有力的方式将代数与解释进行解耦。而且通过模块化的方式，很弹性地更换实现，互相替换。这对于在生产环境中插入测试桩进行测试是有巨大帮助的。所以 free monad 也归入依赖注入模式。
- 纯粹性：可以保持事物的纯粹性与代数性，哪怕组合了 DSL 的整个结构。free monad 提供了整个抽象语法树，就像从 Lisp 宏那里得到的一样多。而且所有事情都在编译期内被类型化及检查。这样也就拥有了可以在其他上下文中重用以及数学推导的结构。代数与解释之间的分割会更加明确清晰。
- 伸缩性：Scala 不支持泛型尾部调用，free monad 的实现使我们可以根据任意

1 4.2.3节中讨论过State monad与StateT。

的复杂度来扩展 DSL，而不用担心撑爆栈。scalaz.Free 的实现会为栈交换堆空间。[1]

5.6 总结

本章主要谈到了在软件工程特别是领域建模中一个很重要的主题。除非学过如何编写模块化的软件，否则模型就是一整块大石头。而且在它漫长的人生中，很难被重构、扩展以及维护。在本章中，我们学到了在Scala里如何将模型分解成多个模块。本章的主要结论如下：

- 什么是模块化？学习了模块的构成部分：代数与实现。还看到如何在发布模块代数的同时保护它的实现。
- 模块的解剖：从个人银行领域中选取了一个案例并学着如何将它拆分到模块。我们设计了代数，研究了领域行为，同时还看到了如何使代数具备组合性，还学习了代数的实现以及如何将它与代数进行解耦。
- 组合性：在模块代数中定义的领域行为需要能够组合，这样才能用代数写出可组合的代码。可以在代数层面实现组合性，这样就可以只在客户应用层面提交实现。
- 模块的物理组织：可以有选择地将模块组织成包并发布它们，这样就只有需要的才会暴露给客户。
- 类型类模式：可以通过一系列不相干的抽象在特定行为上实现多态。我们也看到它所能提供的好处超过了子类型多态。这种模式允许根据事后原理向已存在的类中添加行为。
- 边界上下文：相对于单一模块，边界上下文提供了更大粒度的模块化。我们学习了如何实现边界上下文，以及如何管理多个边界上下文之间的通信。
- *free monad*：这是我们学习的最后一个主题，是模块化中的高级模式，如果首次阅读的话可以跳过它。但对于将计算结构与解释进行分割，该模式可以发挥很大作用。核心思路是在一个封闭的ADT上将计算单元定义为纯数据类型。然后可以用 Free 数据类型来获取一个 monad，在它之上可以构建 DSL。最终得到整个结构并作为一个 AST，可以以后再去解释它。

1 参见Runar Bjarnason所写的*Stackless Scala with Free Monads*（http://blog.higher-order.com/assets/trampolines.pdf）。

6 响应式模型

本章包括

- 介绍响应式领域模型
- 使用 future 和 promise 的响应式 API 设计
- 使用异步消息的边界上下文间的响应式通信
- 流模型以及 Akka Streams 中的实现案例
- 通信的 actor 模型

我们已经学习了很多关于函数式编程的优点，以及当设计一个领域模型时应该如何应用这些概念。本章会首先讨论如何使函数式领域模型具有响应性。我们希望模型能快速地响应用户、保持良好的伸缩性、对错误有良好的弹性，同时使用消息驱动的架构将时间和空间进行解耦[1]。本章覆盖了使用 Scala 建设响应式领域模型的多个方面。可以先从低级别的并发开始，比如线程，再到 Scala 提供的高阶抽象。这里面还包括使用 future 和 promise 的 API 设计、用异步消息实现松耦合的架构、建立流管道，以及最后对 actor 模型的使用也提供了容错与冗余。

1 The Reactive Manifesto，可访问www.reactivemanifesto.org。

图 6.1 显示了本章主题的示意图。这个指南可以帮我们在学习过程中有选择地学习自己感兴趣的主题。

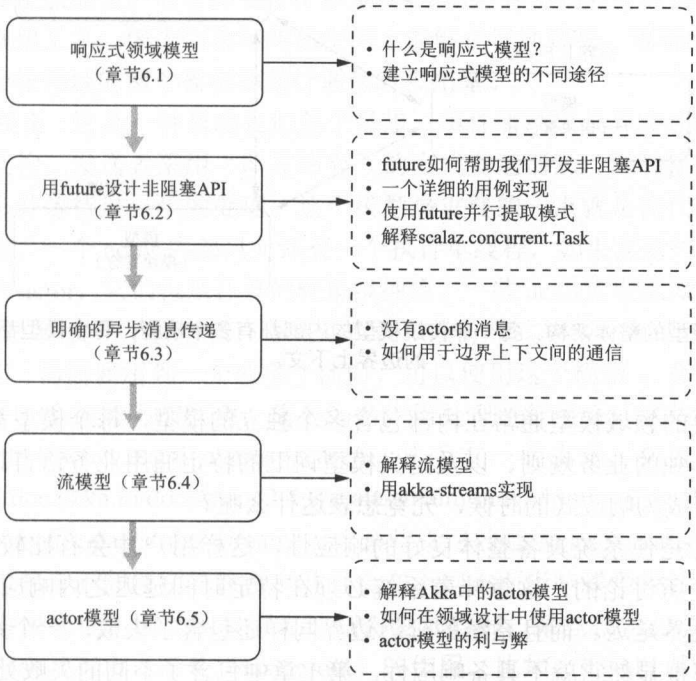


图 6.1 本章学习过程。

学习完本章之后，将知道如何通过合适的抽象使领域模型对用户保持良好的响应性，同时对失败保持弹性。

6.1 响应式领域模型

在第 1 章中已经看过响应式模型的基本概念。第 1 章的图 1.10 展示了响应式模型应该具有的核心属性。但那个讨论主要是由系统设计的泛型原则所驱动。本章会结合函数式领域模型架构将这些思想都串起来。

基于有关模型与边界上下文的讨论，图 6.2 展示了任何中等复杂度的领域模型都可能会具备的整体结构。这当然包括我们一直在讨论的模型：个人银行与投资银行的领域。

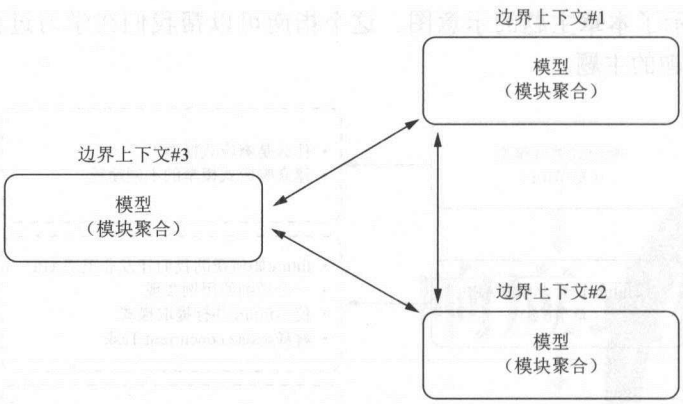


图 6.2 领域模型的整体架构。每个非传统模型的内部都有多个模型，每个模型都被设计为独立的边界上下文。

一个重要的领域模型通常在内部包含多个独立的模型，每个模型都有一系列不同的约束、单独的业务规则、以及定义模型词汇的特定通用业务语言。那么，当我们说希望模型成为响应式的时候，究竟想表达什么呢？

终极目标是使系统具备整体良好的响应性，这样用户才会有比较良好的体验。正如第 1 章中所讨论的，这意味着系统必须在特定时间延迟之内响应用户的请求，这也被称为*边界延迟*。而且系统的延迟边界同样还包含了失败，一个系统如果被失败卡住了，那很显然它就不具备响应性。第 1 章中包含了不同的失败处理策略。

要使模型具备响应性，需要确保构成它的所有模型也都具有响应性。一个模型所需要遵守的响应式架构的所有原则同样适用于构成系统的其他模型。整体系统的响应性取决于组件中响应能力最差的那个。

一个常见的问题是，一个模型内哪些组件需要成为响应式的？是否所有函数都真的需要成为非阻塞的？或者是否所有对象都需要成为用异步消息实现模型行为的 actor？有些学校支持*一路都是 actor* 的原则。但这个设计方式有一些很明显的缺陷，暂时先假设我们能够判断并决定哪些函数是非阻塞的，哪些需要用异步消息的方式来实现。但有一件事是真的：我们需要将响应性作为整体架构的组成部分来考虑。它不是在实现后期才想起来的马后炮。

本章节会简要介绍在领域模型中实现响应式范式的工具。并不是所有工具都适用于我们的用例场景，但有个全面了解还是有帮助的。至少可以对我们做的选择有个判断。在接下来的章节中，将对它们进行深入学习，也将谈到不同的实现技术。

下面清单中的技术可以用来使领域模型成为响应式。

- *future 与 promise*：使用 future，可以为模型设计异步非阻塞的 API，可以使用高阶函数来组合多个 future 并生成一个最终结果。使用 future 会改变设计

API 的方式，我们将看到如何改进第 5 章中设计的某些 API。

- *明确的异步消息传输*：当不同的边界上下文需要交互时，通常建议将这些交互保持松散耦合。更多详细讨论参见 5.4.2 节。异步消息是对这种交互进行建模的理想工具，因为它在时间和空间上能够良好地解耦。围绕消息设计 API，就不会在领域的两个模型间进行通信时被阻塞。
- actor 模型：这是一种帮助我们基于异步、事件驱动以及异步通信设计模型的基本架构。发送者采用“即发即弃”的方式发送消息，同时接收者的信箱接收消息并等待进一步的处理。这个模型最重要的一点就是在代码中不需要处理并发。一个 actor 内部只允许有一个执行单线程，如果需要并发，就要创建大量的 actor，它们也是计算的轻量级抽象。一些 actor实现比如 Akka（http://akka.io）还提供了像容错、位置透明性以及分布式等特性。
- *流模型*：当需要用到一个或多个流时，可以使用这个模型 。在这里数据可能来自于一个连续不断的源，需要为多作用处理进行组合。这些被称为*响应式流*（http:// www.reactive-streams.org），其中有一些实现，包括 Akka Streams（http://doc.akka.io/docs/akka/2.4.4/scala/stream/index.html）和 scalaz 流（https://github.com/scalaz/scalaz-stream）。本章将学习的是 Akka Streams。

每种技术都有不同级别的粒度，并且会在不同方面影响模型设计。某些技术比其他技术更有侵害性，因此需要事先计划得更加仔细。某些技术与 API 设计相关，而且在模型中会具有*由内而外*（inside-out）的作用。比如，把 API 从阻塞式函数调用改变为基于 future 的非阻塞式调用时，不仅在 API 层面会产生冲突，还会进一步延伸到应用层。其他像监控与容错是*由外而内*（outside-in）的，它们需要从模型的外围进行注入。计算模型，如 actor 模型，往往会对设计产生影响。所有的交互都通过消息进行传递而不是函数调用。比较少类型的计算会采用即发即弃的 actor 消息处理模型。因此，将这种模型用作终端用户交互点时，推导代码就会变得非常困难。接下来就会学习这些内容，并发现领域模型内最有意义的场景。使模型成为响应式的最常见的方式就是用 future 和 promise 设计 API，我们将从此开始。

6.2 使用future的非阻塞API设计

成为响应式的一个核心原则是确保设计不会创建任何形式的争夺或瓶颈从而妨碍系统的运行。想象一下，如果领域服务发布的 API 阻塞了对底层数据库的调用以及用户交互的核心线程，那么这个 API 设计很明显是有问题的：阻塞中断了流程，进而导致了很差的用户体验。

本节的例子来自于第 5 章中设计的模块 API，同时使用 future 对它们进行改造，使其变为非阻塞的异步。如本章前面所说，需要判断清楚哪些要调用的 API 需要成为非阻塞的，而不是一股脑使整个模型全都变成非阻塞的。在这个讨论的最后，将会看到为什么将这些 API 改造成非阻塞是一个合理的策略。

让我们将 5.2.1 节中讨论的模块代数来作为案例。我们在清单 6.1 中再次看到这个代数。

清单 6.1 模块 Account Service 的代数

```
trait AccountService[Account, Amount, Balance] {
  type Valid[A] = NonEmptyList[String] \/ A
  type AccountOperation[A] = Kleisli[Valid, AccountRepository, A]

  def open(no: String, name: String, rate: Option[BigDecimal],
    openingDate: Option[Date], accountType: AccountType):
      AccountOperation[Account]

  def close(no: String, closeDate: Option[Date]): AccountOperation[Account]
  def debit(no: String, amount: Amount): AccountOperation[Account]
  def credit(no: String, amount: Amount): AccountOperation[Account]
  def balance(no: String): AccountOperation[Balance]

  def transfer(from: String, to: String, amount: Amount) = //..
}
```

模块定义——领域友好的命名以及类型参数化

拥有领域友好命名的操作定义

每个模块 API 均返回一个 Kleisli，它们可以很好地进行组合并实现更强大的功能。在设计这个模块时，我们忽略了一个非常重要的方面：与底层 AccountRepository[1] 的交互以及在应用级别可能导致的边界延迟。如果这些操作需要花费很长时间，则执行的主线程将会被阻塞进而导致糟糕的用户体验。在这个场景中需要 API 有足够的弹性，以便用户感知到的响应不会被当前系统负载所影响。也正如在第 1 章中所讨论的，这是基于 future 计算的本质：通过返回 Future 使操作变成非阻塞的。我们要如何修改呢？

6.2.1 异步作为堆叠作用

我们需要改变 API 以及它们的实现，同时通过 API 的类型表达响应式目的。当前代数中，Kleisli 有助于咖喱化 AccountRepository 及多作用函数应用。我们希望在新版本中也保持同样的方式。关键部分就是分离 scalaz.\/，它是计算中返回值或错误的和类型。[2] 需要在计算分离之前在某个地方插入 Future。

1 通常情况下它是一个企业级数据库。

2 2.4.1节已经讨论过和类型与乘积类型。

这是一个用特定方式捆绑两个作用（分离（\/）与 Future）的练习，这样 Future 就立即可用，而分离只有当计算完成后才可用。幸运的是两个作用都是 monad，因此可以用 *monad 转换器*来粘合它们。我们马上就可以看到具体实现，不过在此之前先看一下什么是 monad 转换器。

Monad 转换器

在第 4 章中，做单子绑定（Scala 里的 flatMap）时，一个步骤的执行依赖于上一步的成功或失败。因此单子作用并不能保持计算的形态。相关内容可以复习 4.2.3 节。也正因为这个原因，我们说 *monad 不能用 applicative* 那种通用方式进行组合。

为了创建一个合成的 monad，可以使用 monad 转换器。这个技术就是将 monad 叠加到一起，给一个转换后的 monad。然后就可以在合作的 monad 上使用表达式并从中提取出值。这不是泛型技术，它只对 monad 集合有效。

我们看一下下面的例子，它对从存储中查询领域对象的返回类型进行了建模。

```
type Response[A] = String \/ Option[A]
```

这里的 Response 是一个 monad，因为在 Scalaz 中 \/ 是一个 monad。我们可以用普通的右偏 Either 类型（也就是 \/），从右侧取得值。但这里右侧的值是一个 Option。为了达到查询返回的计数，需要在 \/ 内处理另外一个 monad。那么如何从 Response 中取得着重的 count 值呢？这里首先尝试一层层地剥离单子层：

```
val count: Response[Int] = some(10).right
for {
  maybeCount <- count
} yield {
  for {
    c <- maybeCount
  } yield c
}
```

将一个 Option 提到 V 里，将得到一个 Response。认真看一下 Response 类型声明并说服自己真实情况就是如此。

使用 c 的代码。

这种方式的问题是它不能测量评估。第二层的 for 表达式嵌套在第一层中。一旦 Response 内叠加了越来越多的作用，将遇到更多表达式的嵌套。理想情况是，如果能够使用 \/ monad 并给右侧的值添加处理可选项的作用（也就是说在 \/ 的顶部粘合一个 Option，这样就可以把组合当成一个单一的 monad 来处理）。这就像将两个 monad 捆绑成一个一样。这也正是 monad 转换器所做的事。下面是采用这种粘合技术的例子：

```
import scalaz.{ \/, OptionT }
type Error[A] = String \/ A
type Response[A] = OptionT[Error, A]
```

这里的 OptionT 就是一个 monad 转换器，它给我们一个单一 monad，在底层 monad（在这个案例中就是 \/）的顶部增加了一层 Option。现在可以只用一个 for 表达式就取到 count 的值。下面是使用 OptionT 的代码：

```
import scalaz.syntax.monad._
val count: Response[Int] = 10.point[Response]
for {
  c <- count
} yield (())
```

将计算提到 monad 转换器 OptionT 中。

可以使用 for 表达式，因为转换同样也是一个 monad。

在这里使用 c.c 是一个 Int。

从用户的角度来说，这看起来更加清爽，因为消除了 for 表达式内额外的嵌套步骤。然后在将 Future 和 Either 叠加到一个 monad 的实现中运用这个技术，并且实现异步作用，就像在 6.2.1 节开始所讨论的那样。

叠加 Future 与 Either

当前模型用的是转换器 EitherT。这是一个很重要的概念；大家看到使用 Future[NonEmptyList[String] \/ A] 的问题了吗？这跟我们讨论 OptionT 时遇到的是同一个问题。要通过嵌套的 for 表达式做额外的间接跳转，剥离计算的多个单子层。

与此相反，monad 转换器 EitherT 提供一个单独的 monad，它把 Future 的作用叠加在 Either 内部。这个建模技巧的基本思路是：使类型尽可能地接近我们要实现的作用。

当我们将类型用作模型定义的固有部分时，作用的粘合就相当于一个额外的类型，它被插入到一个已经存在的叠加内。这里将 Future 作为一个作用来建模响应式 API，并且给 monad 转换器增加一个层。恰当的类型定义可以将这些改动限制在定义转换器的单一类型别名。只需要将别名 Valid 改为 type Valid[A] = EitherT[Future, NonEmptyList[String], A]，其他代数中的内容与第 5 章清单 5.1 中内容保持一致即可。图 6.3 展示了这个技巧。

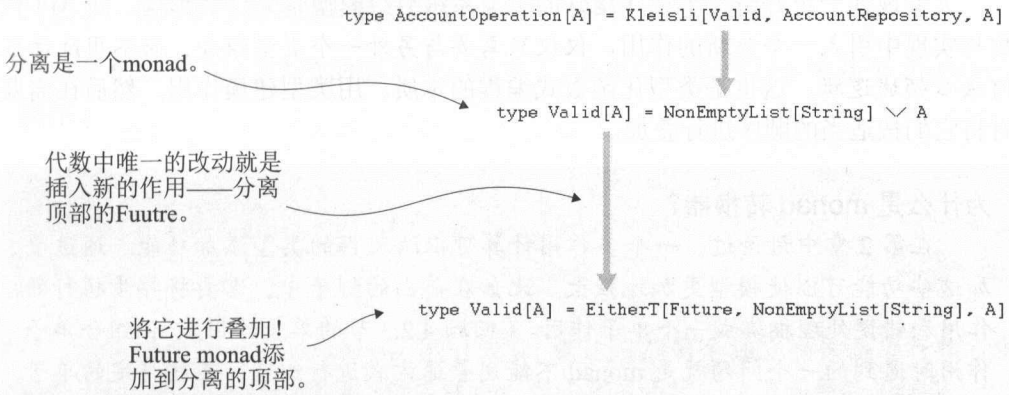


图 6.3 类型驱动的作用组合：将异步与非阻塞当成作用并通过 monad 转换器对其进行建模。代数中唯一的改动就是别名为 Valid 的类型。代数其他部分保持不变。

6.2.2 基于 monad 转换器的实现

让我们来看一下在模块 API 实现中所需要的改动。要用转换器将两个作用叠加到一起，同时核心领域逻辑与第 5 章清单 5.2 中的保持一致。清单 6.2 展示了这个例子。

清单 6.2 AccountService 解释程序

```
package domain
package service
package interpreter                          // 与清单 5.2 相同

class AccountServiceInterpreter extends
  AccountService[Account, Amount, Balance] {

  def open(no: String,
    name: String,
    rate: Option[BigDecimal],
    openingDate: Option[Date],
    accountType: AccountType) =
    kleisli[Valid, AccountRepository, Account] {
      (repo: AccountRepository) =>

        EitherT {                            // 用 monad 转换器 EitherT
          Future {                           // 叠加 Either 和 Future

          }                                  // 与清单 5.2
        }                                    // 相同
  }
  //.. other operations
}
```

大家现在一定好奇一个设计良好的类型系统会给模型带来什么影响。向 API 代数与实现中引入一个全新的作用，仅仅只需要与另外一个类型组合，而不用改动任何核心领域逻辑。这也是类型化函数式编程的本质。用类型建模作用，然后在需要时将它们按适当的顺序进行叠加。

为什么是 monad 转换器？

在第 2 章中讨论过，一个多作用计算可以给现存的类型添加功能。通过叠加这些功能可以使模型更加地强大。比如在前面的例子中，需要将异步执行的作用和错误处理捆绑成一个单子作用。（回顾 4.2.3 节的单子作用）。但组合单子作用时遇到的一个问题就是 monad 不能用普通方式进行组合。某些特定的单子可能可以组合，但这种组合下的 monad 不是封闭的。

monad 转换器闪亮登场。通过转换器，可以将一个 monad 的作用加到另一个 monad 的顶层，这样的结果就是一个 monad 组合了两者的作用。要注意叠加的顺序。在创建类型的 monad 转换器时，需要明确顺序。这样做的好处就是可以随心所欲地组合作用，同时依然只使用一个 for 表达式来遍历作用的堆叠。这部分可以仔细看一下本章中所使用的案例。

练习 6.1 处理作用（代数）

在第 4 章中 4.4 节中描述的一个完整案例，就是如何用类型的代数和模式来演进一个 API。最后得到了一个 `tradeGeneration` 函数的实现，它从客户订单和市场执行生成交易。同时不需要了解任何底层类型的具体实现。清单 4.10 展示了实现的最终函数。

我们注意到，用来组成 `tradeGeneration` 实现的每个函数都不会处理在其执行过程中可能产生的错误。比如说，`allocate(as: List[Account])` 获取 `Execution` 并生成一个 `Trade` 的 `List`。如果发生异常，这个函数不会返回给调用者任何关于异常的信息，而异常可能会发生在 `allocate` 的执行过程中。

每个函数已经以 `List` 的方式处理了一个作用。给定一个客户账户的列表和一个执行，`allocate` 返回一个交易的 `List`。如果需要处理错误，一个方法是将异常当作用来处理，将其和已经存在的作用（`List`）进行叠加。我们不能抛出异常（函数式编程中的禁忌），否则就违反了引用透明的原则。

在这个练习中，我们要实现整个用例以便使每个函数都处理异常并返回错误消息给调用者。控制流将自动进行调整。如果执行分配失败，就不会生成任何交易。如果订单执行失败，就不会生产任何执行。

提示：使用一个合适的表示方式来处理异常，并用 monad 转换器将其和 `List` 叠加到一起。在 Scalaz 中查阅已存在的抽象。

6.2.3 用并行存取降低延迟——一种响应式模式

在6.2.1节和6.2.2节的例子中，组合了AccountService中基于Future的操作：

```
for {
      _ <- open(..)
      _ <- credit(..)
      d <- debit(..)
    } yield d
```

组合操作是一个Future，但组成for表达式的每个独立操作是串行执行的。要知道组合操作不会阻塞执行的主线程，当它完成后我们会得到最后的结果。这确实是执行之前用例的有效方式，将独立操作并行执行显然会导致不必要的不确定性。

但在某些场景下，将颗粒状的操作用并行方式执行会让我们从中受益，第1章在讨论使用Future的事件驱动编程时描述的就是这样的场景。在本节中，会更深入地了解实现细节。在这里将看到设计的过程，在本书的代码库中可以获取完整的实现。

比如说银行给消费者提供了一个投资组合服务，我们为用户的投资组合创建了一个领域模型，然后为投资组合报表设计了模块API。下面的贴士中简要地介绍了投资组合报告应该包含哪些部分。

投资组合服务——什么是投资组合？

客户投资组合是指客户在银行的资产详细信息。它包括所有的货币资产，比如普通股、固定收入以及其他证券资产：

- ISIN代码，证券的国际代码
- 持有数量
- 当前市场价值

客户可以通过他们的投资组合报告快速了解自身的整体财务状况。

我们有一个用户账户的简单模型和一个对投资组合实体建模的抽象。清单6.3到6.5描述了Portfolio、Instrument和Account三者的领域模型。[1]Instrument是一个组成所有金融交易基础的实体。如果不熟悉金融交易，下面的贴士做了一个概要介绍。

1 所有领域模型都被简化以便于解释。

金融证券

金融证券是一个有特定特征的资本单位，它也可以在市场上进行交易。市场中的证券交易允许资本在投资者之间高效地流动。

本章包含以下证券类型：

- 货币（CCY），这也是市场中最简单的证券。货币有不同的代码，比如USD（美元）或EUR（欧元）。
- 普通股（EQ），也就是可以持有的股票。
- 固定收入（FI），通常由债券组成，它们定期产生收益，并在最终到期后返还本金。

清单 6.3 Portfolio 领域模型

```
package model
import java.util.Date
import common._

case class Balance(ins: Instrument, holding: Amount,
  marketValue: Amount)

sealed trait Portfolio {
  def accountNo: String
  def asOf: Date
  def items: Seq[Balance]
  def totalMarketValue: Amount =
    items.foldLeft(BigDecimal(0d)) { (acc, i) => acc + i.marketValue }
}

case class CustomerPortfolio(accountNo: String, asOf: Date,
  items: Seq[Balance]) extends Portfolio
```

Balance 是一个值对象。考虑到优化，可以在这里保留一个证券 ID 并为它查找存储。

totalMarketValue 是一个计算字段。

清单 6.4 Instrument 领域模型

```
package model

import java.util.Date
import common._

sealed trait InstrumentType

case object CCY extends InstrumentType
case object EQ extends InstrumentType
case object FI extends InstrumentType

sealed trait Instrument {
  def instrumentType: InstrumentType
}
```

在这里定义 3 个证券类型。

一个极度简化的 Instrument 模型。

```
case class Equity(isin: String, name: String, issueDate: Date,
  faceValue: Amount) extends Instrument {
  final val instrumentType = EQ
}

case class FixedIncome(isin: String, name: String,
  issueDate: Date, maturityDate: Option[Date],
  nominal: Amount) extends Instrument {
  final val instrumentType = FI
}

case class Currency(isin: String) extends Instrument {
  final val instrumentType = CCY
}
```

清单 6.5 Account 领域模型

```
package model

import java.util.{ Date, Calendar }
import scalaz._
import Scalaz._

object common {
  type Amount = BigDecimal
  def today = Calendar.getInstance.getTime
}
import common._

case class Account(no: String, name: String,
  dateOfOpen: Option[Date] = today.some, dateOfClose: Option[Date])
```

在本节中，会看到生成用户完整投资组合的服务案例。完整的投资组合包括货币、普通股以及固定收入。因为这些是证券的不同类型，所以投资组合的信息需要从不同的系统中进行提取。因此，它们可以并行执行并最终汇总到一起形成完整的用户投资组合。图 6.4 描述了这个场景。

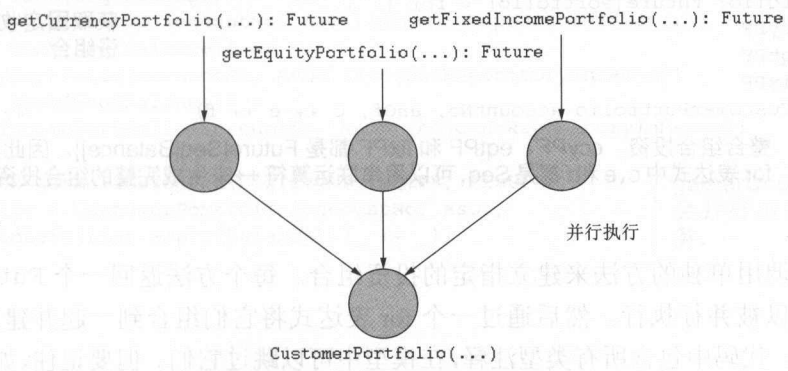


图 6.4 并行执行提取方法，并最终组合成完整的投资组合。

清单 6.6 展示了模块 PortfolioService 的服务框架。

清单 6.6 投资组合报告服务

和前面一样，用 Kleisli 来注入存储。

```
trait PortfolioService {
  type PFOperation[A] = Kleisli[Future, AccountRepository, Seq[A]]

  def getCurrencyPortfolio(no: String, asOf: Date): PFOperation[Balance]
  def getEquityPortfolio(no: String, asOf: Date): PFOperation[Balance]
  def getFixedIncomePortfolio(no: String, asOf: Date): PFOperation[Balance]
}
```

每个方法返回一个特定的投资组合，它将提取指定类型证券的余额列表。比如，一个用户可能有多个普通股，而 getEquityPortfolio 方法将返回所有余额的列表。

本章没有包含该服务的具体实现，完整代码可以在代码库中找到。让我们聚焦在如何用这些服务建立一个非阻塞响应式的投资组合计算功能。清单 6.7 给出了一个实现样例。

清单 6.7 建立完整的投资组合

获取货币投资组合 ❶

获取普通股投资组合 ❷

获取固定收入投资组合 ❸

```
import PortfolioService._

val accountNo = //..
val asOf = //..

val ccyPF: Future[Seq[Balance]] =
  getCurrencyPortfolio(accountNo, asOf)(AccountRepositoryInMemory)
val eqtPF: Future[Seq[Balance]] =
  getEquityPortfolio(accountNo, asOf)(AccountRepositoryInMemory)
val fixPF: Future[Seq[Balance]] =
  getFixedIncomePortfolio(accountNo, asOf)(AccountRepositoryInMemory)

val portfolio: Future[Portfolio] = for {
  c <- ccyPF
  e <- eqtPF
  f <- fixPF
} yield CustomerPortfolio(accountNo, asOf, c ++ e ++ f)
```

整合组合投资。ccyPF、eqtPF 和 fixPF 都是 Future[Seq[Balance]]。因此在 for 表达式中 c、e 和 f 都是 Seq，可以用串联运算符 ++ 来生成完整的组合投资。

首先调用单独的方法来建立指定的投资组合。每个方法返回一个 Future，同时它们可以被并行执行。然后通过一个 for 表达式将它们组合到一起并建立最终的投资组合。代码中包含所有类型注释，在模型中可以跳过它们。但要记住，如果把 ❶、

❷、❸ 的 val 创建放到 for 表达式里，那么这个实现就不再是并行非阻塞的了。

6.2.4 使用 scalaz.concurrent.Task 作为响应式构造

第 5 章提供了另外一种模块以及组合的实现，就是基于 free monad。（参见 5.5.2 节）free monad 帮助建立递归数据类型的抽象语法树，而且不用为它们应用解释程序。我们可以完全按照自己的意愿来解释这个树。

free monad 将计算编码为抽象语法树，解释程序需要成为响应式的，才能得到一个响应式的实现。这也正是 5.5.4 节的 AccountRepoInterpreter 中所做的。不过在这里我们不用 Future，用另外一个构造，scalaz.concurrent.Task：

```
import scalaz.concurrent.Task
trait AccountRepoInterpreter {
  def apply[A](action: AccountRepo[A]): Task[A]
}
```

Task 通过一个类型类将异步执行抽象成一个作用，Nondeterminism。下面的贴士会提供更多关于 Task 抽象的信息以及它与 Future 的区别。因为解释程序返回的是 Task，所以实现就已经是响应式的了。

针对投资组合的例子，可以使 PortfolioService 返回一个 Task 而不是 Future，这样生成完整投资组合的应用代码就会变得更加简单。清单 6.8 展示了这一点。

清单 6.8 对 PortfolioService 使用 Task

```
import PortfolioService._

val accountNo = "a-123"
val asOf = today

val ccyPF: Task[Seq[Balance]] =
  getCurrencyPortfolio(accountNo, asOf)(AccountRepositoryInMemory)
val eqtPF: Task[Seq[Balance]] =
  getEquityPortfolio(accountNo, asOf)(AccountRepositoryInMemory)
val fixPF: Task[Seq[Balance]] =
  getFixedIncomePortfolio(accountNo, asOf)(AccountRepositoryInMemory)

val r = Task.gatherUnordered(Seq(ccyPF, eqtPF, fixPF))      <-- gatherUnordered 会并行地执行任务。
val portfolio = CustomerPortfolio(accountNo, asOf,
  r.run.foldLeft(List.empty[Balance])(_ ++ _))
```

scala.concurrent.Future 与 scalaz.concurrent.Task

可以看到，Future（来自 Scala 标准库）和 Task（来自 scalaz.concurrent）都可以使模型成为响应式，并且都可以使 API 以异步和非阻塞的方式运行。但这两个抽象有本质的区别。要选一个来用的话，需要有自己的判断：

- Future 是一个标准库中的抽象，所以不用依赖任何第三方的库就可以使用 Future。
- Future 建模了运行计算。它对如何建立计算以及如何单步运行计算进行了抽象，就 API 设计的工程角度来说这还是有点不够清晰。而另一方面，Task 在如何形成计算的描述与计算的执行之间有非常明确的界限。可以通过明确的上下文，一个名为 Nondeterminism 的类型类，来控制不确定性的程度。[1] 这个上下文允许提供所要建模的不确定性的确定行为。Task 是一个 monad，因此执行一个 Task 时，默认的策略就是像 monad 那样 *bind*。但通过明确定义的执行上下文如 Nondeterminism，可以改变执行策略。如果觉得有太多事要做，那就不要再做一系列的 bind，并行地执行它们，并返回首先完成的那个。
- 基于 Future 的 API 可以通过提供 Future 的一个 Monad 实例而变成单子化。这是设计响应式 API 时通常采用的策略。但也正如 6.2.1 节所述，如果想要在现成的单子化 API 顶部叠加一个 Future，就需要用到 monad 转换器。monad 转换器不是很直观，特别是在 Scala 中，普遍认为它会产生难以理解的代码结构。Task 提供了大量不需要 monad 转换器的组合器，因此设计和使用基于 Task 的响应式 API 通常会更加简单。
- Task 是一种可以工作在堆栈上的“蹦床”抽象。这在处理交叉并发结构时成为一个很重要的优势。可以看一下 Runar Bjarnason 所写的 *Stackless Scala with Free Monads*（http://blog.higher- order.com/assets/trampolines.pdf），了解蹦床如何帮我们设计深层嵌套的结构。

本次讨论的主要结论如下。

- **由内而外的响应**：这里所讨论的响应式特性与 API 的设计和实现相关，它从模型的核心向外流向用户的应用。
- **类型分层**：我们讨论的实现技巧是完全无损的。利用 monad 转换器的优势，可以将 Future 作为一个作用黏合到已存在的 monad 顶部，从而插入响应式特性。
- **异步与非阻塞**：非阻塞的 API 可以使主线程保持对其他活动的响应。不用再等待一个阻塞式的 API 从托管在远程数据中心的数据库中返回一堆数据。

- 自由意味着更多优势：如果有一个基于 free monad 的架构，那么 free monad 就彻底与添加的所有作用解耦。唯一要做的就只是改动解释程序。

本书代码库中包含响应式版本的 API 及实现的完整代码。

练习 6.2 Future 与 scalaz.concurrent.Task

清单 6.1 中的 AccountService 使用 monad 转换器将 Future 叠加进单子 API。请使用 scala.concurrent.Task 替代 Future 重新实现该服务。可以参考前文比较 Future 和 Task 的讨论内容。

贴士：不需要使用 monad 转换器。API 会比基于 Future 的更加简单。

6.3 明确的异步消息传递

当谈论响应式系统时，首先想到的是它们如何解决系统各部分之间的耦合问题。经常会听到人们争论，为了保持系统的响应性，需要使各组件尽可能地保持松耦合。这是什么意思？如何知道哪些组件需要保持松耦合？如何保证它们松耦合？

这里需要理解的一个很重要的概念就是异步边界。当两个在空间和时间上解耦的实体需要进行相互通信时，这个通信必须是异步的。同步调用要么长时间阻塞，要么失败，因为两者之间缺乏关于通信所达成一致的协议。这是异步边界的一个例子，在这里需要在两者之间异步地传递信息。在现实世界中存在相当多的异步边界。我们会遇到下列的异步边界：

- 应用或边界上下文
- 线程
- 角色
- 多核 CPU
- 网络节点

即使将两个边界上下文部署在同一台机器里，也需要考虑跨异步边界处理数据的移动。本节会介绍如何用异步消息传递来设计领域模型的案例。这里暂不会包含 actor 模型，我们会用独立的章节来讨论它。

异步消息的典型使用场景是多个边界上下文之间通信的建模。一个边界上下文是典型的运行在其自身空间和时间领域内的独立应用。但边界上下文之间依然需要进行交互，虽然可能是不定期的交互。考虑第 5 章中的两个边界上下文：账户管理处理管理用户账户的在线请求，而报告与分析负责对系统中所有账户生成报告以及 MIS 信息。

我们不能在两个上下文之间使用同步通信，理由如下：

- 它们可能无法配合。
- 它们可能根据不同的时间限制被建模为服务（比如，报告只作为夜间任务执行，而账户管理却是 7 天 ×24 小时的服务）。
- 它们具有不同的数据模型以及不同的通用语言（比如，Account 在两个上下文中都是有效存在，但却有不同的属性集合）。

图 6.5 描述了两个边界上下文通信的案例。在这个场景中，账户管理上下文生成用户上下文所需信息的事件或消息。而报告是顺从型的上下文，这意味着它必须遵守账户管理所生成的消息格式或协议。[1]

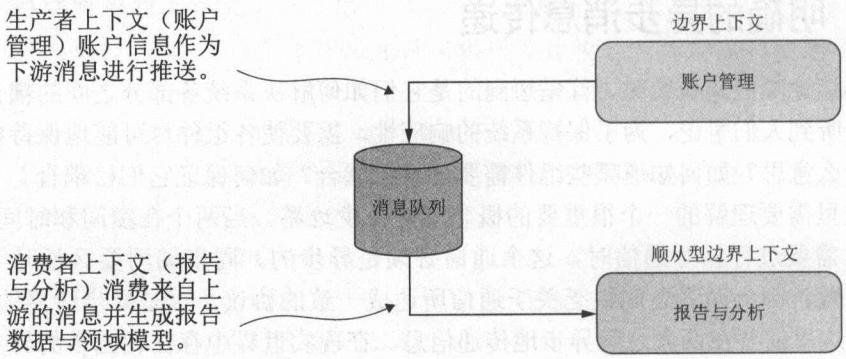


图 6.5 时间与空间分离的边界上下文，它们通过异步消息进行通信。

消息传递是通过一个消息队列在进行。生产者上下文（账户管理）将消息推送进队列，消费者（报告）拉取它所需的消息。这个消息队列可以用诸如 Kafka（http://kafka.apache.org）或 RabbitMQ（www.rabbitmq.com）实现，或用任何提供队列语义的存储来建模。

所以，明确的异步消息传递的基本结论如下。

- *异步边界*：用这个策略来跨异步边界传递信息。记住，甚至在同一台机器内也依然存在跨线程或跨 CPU 内核的异步边界。
- *时间与空间的解耦*：这种编程模型提供了时间与空间上的解耦，在两个弱连接的系统间使用这种策略来进行通信（比如，边界上下文之间的通信）。

1 参见Eric Evans所著的*See Domain-Driven Design: Tackling Complexity in the Heart of Software*（Addison-Wesley Profes sional, 2003年）。

6.4 流模式

正如前面章节中所学习的内容，通过明细的消息传递使异步边界之间通信变得非常便捷，而且这也是信息交换最合适的模型。前提是通信不是非常密集，而且系统在时间和空间上是解耦的。但是经常会遇到另外一种情况：模型的一个组件生成了一个数据流，这个数据流在被多个下游组件消费前需要通过不同方式进行处理和传输。本节将提供一些处理数据交互的流模型的方法，帮助我们保持系统的响应性。

让我们先从设计的角度看一下这个模型所带来的挑战有哪些。

- *信息的连续流动*：在很多场景中，流是无限生成的（比如，监控系统中的股票指数、市场价格、汇率以及时间序列数据连续不断地流进系统）。如何在不给系统带来巨大负载的情况下处理它们？
- *孤立观察*：对于流数据，只能看到一次数据。数据必须在线处理，而且不能从流中删除任何数据。
- *存储问题*：有一个持续的数据流时，不能指望将它们全部都存储下来之后再做处理。再强调一次，处理必须是在线的，而且是最小延迟的。
- *后端压力处理*：在大部分情况下，会看到数据的消费者通常跟不上数据的生成速度。如何管理这种不匹配？

记住这些问题，这里有一些建议的途径可以解决它们。不是每个流处理框架都会实现所有这些特性。但作为一个模型设计者，拥有整体的视角总是一件好事。以下是一个可扩展的流处理 API 应该具备的一些特质。

- *异步*：为了保证计算资源的优化利用，处理必须是异步的，这些计算资源可以是多个协作网络节点，或者是一台机器里的多个 CPU 核心。
- *非阻塞*：要求能确保边界延迟，类似于 6.2 节中讨论的基于 future 的 API。
- *弹性处理后端压力*：当生产者推送数据给消费者，同时消费者的处理速度低于生产速度时，这一点就变得很重要了。
- *可组合*：这是所有 API 都应该具备的特质，也有利于更好地建模。

6.4.1 一个案例

接下来不再过多讨论设计流模型 API 背后的理论，让我们来看一个领域中的例子，以及如何用一个流处理框架来对它进行建模。

在这里例子中，将考虑银行交易的下游处理。我们有一个资源用来接收来自一个或多个系统的银行交易流。模型需要把这个流拆分成多个子流，将它们按账户编号分组，对每个子流选择性地转换，并最终为每个子流计算净交易值。图 6.6 描绘了这个场景。

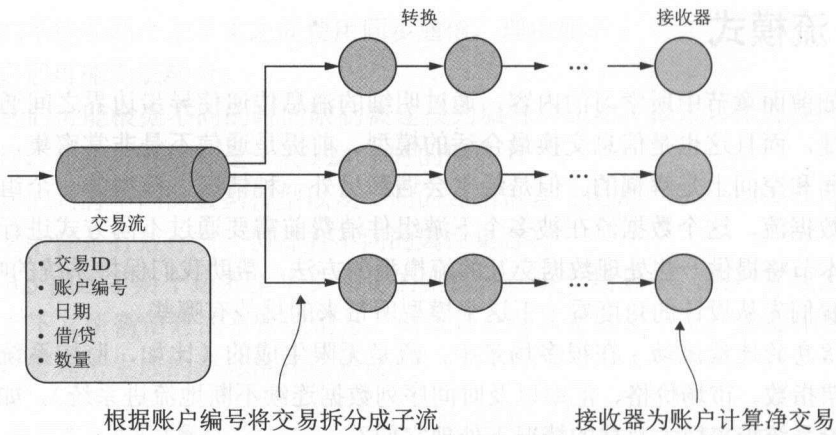


图 6.6 银行交易的处理管道。来自上游系统的交易流进入系统，根据账户分组并拆分成子流，在进行一些转换和校验之后进入接收器。

我们将使用 Akka Streams 作为实现框架。这个案例的标准化基于 http://reactive-streams.org 所发布的规范。很多实现都基于这个规范，最终应该在所有响应式流实现上提供无缝的交互操作。

作为实现框架的一部分，Akka Streams 定义了以下抽象，用于建立流处理管道。

- *Source*：这是管道开始的地方。`Source[+Out, +Mat]` 从输入获取数据，同时向一个单一输出 `Out` 写入数据。
- *Sink*：这是管道终点。一个 `Sink[+In, +Mat]` 有一个单一输入 `In` 被写入数据。
- *Flow*：这是进行数据转换的基本抽象。一个 `Flow[-In, +Out, +Mat]` 有一个输入 `In` 和一个输出 `Out`。`Mat` 是指在列表最后所描述的实现器。所以绝大部分的管道由一个 `Source`、多个 `Flow`（也可能是 0）以及一个 `Sink` 组成（`Source->Flow*->Sink`）。
- *RunnableGraph*：（`Source->Flow*->Sink`）的整个拓扑形成了 Akka Streams 里的 `RunnableGraph`。在 `Source` 和 `Sink` 之间的多个 `Flow` 结构组成了结合点，它们对流的扇入（fan-in）或扇出（fan-out）点进行建模。一个结合点可以是 `Broadcast` 的类型，即扇出流控制。或者也可以是一个 `Merge` 类型，即扇入模型。同时整个拓扑在 `Source` 和 `Sink` 之间是闭合相连的，随时都可以被执行，因此也被称为 `RunnableGraph`。在后面的章节中会看到执行中的 `RunnableGraph`。
- *Materializer*：即实现器，它定义了转换如何变成异步处理。比如，`Actor-Materializer` 通过 actor 完成这个转换。

回到我们的案例中，下面是如何在模型中定义 Transaction：

```
sealed trait TransactionType
case object Debit extends TransactionType
case object Credit extends TransactionType
case class Transaction(id: String, accountNo: String,
  debitCredit: TransactionType, amount: Amount, date: Date = today)
```

接下来是实现处理管道，这里将使用 Akka Streams 提供的抽象。

步骤 1：建立 Source

处理管道从建立 Source 开始，它抽象了所有流处理步骤并包含一个单一输出。从 Source 开始，通过对 Source 输出的数据添加转换来建立 Flow，并最终将流输入给一个 Sink：

```
import akka.actor.ActorSystem
import akka.stream.scaladsl._
import akka.stream._

object TransactionPipeline {
  implicit val as = ActorSystem()
  implicit val ec = as.dispatcher
  val settings = ActorMaterializerSettings(as)
  implicit val mat = ActorMaterializer(settings)

  val transactions: Source[Transaction, akka.NotUsed] =
    Source.fromFuture(allTransactions).mapConcat(identity)

  //..
}
```

为使用 Scala DSL 和流的导入。

建立 actor 系统与实现器。

建立获取交易流的 Source。注意 akka.NotUsed 采用了泛型类型，在这里实际值并不重要。把它当作 Scala 里的 Unit。

所有的输入和代码都应该充分自注释。最后一步建立开始管道的 Source。在这里例子中，Source 从 allTransactions 函数获取输入，该函数返回 Future，包含所有进入系统的交易：

```
def allTransactions(implicit ec: ExecutionContext):
  Future[Seq[Transaction]] = Future {
  //..
}
```

步骤 2：将流拆分成子流

建立 Source 之后，根据图 6.5 中的步骤，需要根据账户编号拆分流。所以将每个账户编号拆分流，确切的数字由下面的 groupBy 操作决定。在拆分前，可以因为校验或其他任何领域特定的约束而插入转换：

```
transactions.map(validate)
            .groupBy(MaxGroupCount, _.accountNo)
```

映射 Source 以及校验交易

根据账户编号分组交易，这样就拥有了每个账户的子流。MaxGroupCount 是分组的最大数目，一旦超过就会报错。

步骤 3：在 Sink 结束

作为例子的最后一步，现在需要基于借 / 贷标签计算每个账户的所有净交易，并在一个 Sink 结束流。以下代码以完全函数式的方式按顺序地执行了所有步骤，同时用到了 Akka Streams 提供的丰富的组合器。这个代码是自注释的，如果想了解更多细节，请看代码后面的解释。

```
val netTxn: RunnableGraph[Future[Transaction]] =
  transactions.map(validate)
              .groupBy(MaxGroupCount, _.accountNo)
              .fold(TransactionMonoid.zero)(_ |+| _)
              .mergeSubstreams
              .toMat(txnSink)(Keep.right)
```

❶ fold 子流

❷ 一旦完成净值计算就合并子流

❸ 在 Sink 终结

在有了多个子流之后（每个账户一个）、得到一个可运行的 RunnableGraph 之前需要先完成以下内容。再次注意流的 API 是如何将计算和执行解耦的：

- 对每个子流进行净值计算。对 groupBy 返回的 Subflow 执行 fold❶。fold[1] 使用 TransactionMonoid 来对每个子流累积交易并汇总数额。[2]
- 完成净值计算之后，合并子流。❷
- 最终实现为一个 Sink，得到 RunnableGraph。❸

现在得到了 RunnableGraph，可以运行它并得到结果。在之前的代码中，txnSink 是一个 Sink，在这里具体化了计算。Akka Streams 为具体化定义的计算使用了一系列的 actor。对于这个例子，可以将 Sink 定义为在控制台打印输出：

```
val txnSink: Sink[Transaction, Future[akka.Done]] = Sink.foreach(println)
```

代码库中有这个例子的完整可运行版本。

1 本例中，子流上的fold是串行执行。如果fold内的每个独立子流操作都比较慢，可以在fold之后加上.async使其并行执行。

2 如果需要回顾monoid，可参见4.1节。

步骤 4：运行计算

现在有了等待运行的 RunnableGraph。这里有个运行的例子，管道从输入流开始读取数据并在 sink 生成输出：

```
netTxn.run().foreach(println)
```

6.4.2 领域管道图

让我们看一个稍微复杂一点的流模型例子，在这里领域行为的调用序列可以被建模为图形。像往常一样，我们将区分图形的构建阶段和执行阶段的差异。作为一个建模者，我们将聚焦在领域行为上，以及它们如何与管道交互。后台框架 Akka Streams 在执行与图形模型管理的背后做了大量繁重的工作。

图 6.7 描绘了即将建模的管道。

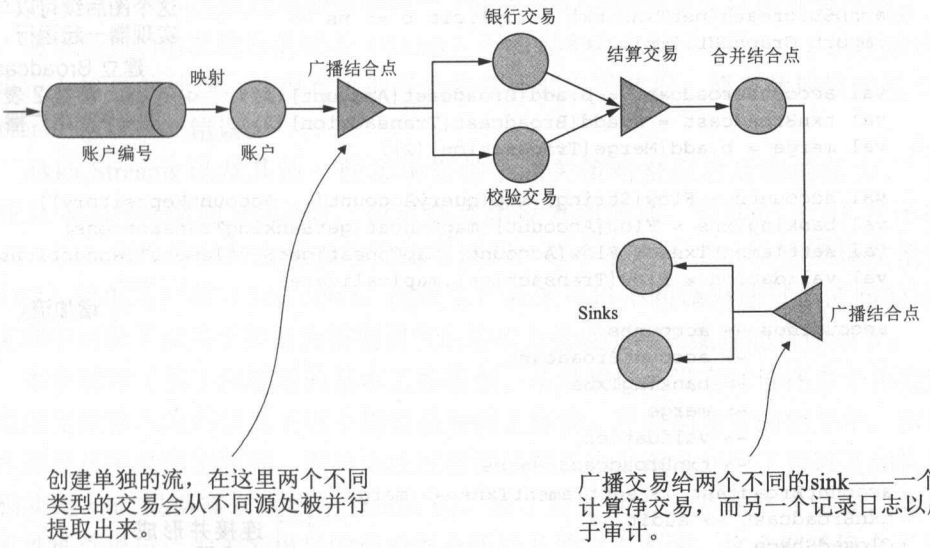


图 6.7 在 Akka Streams 里活动的管道建模为图形。图中文字描述了管道的不同阶段。

以下是管道在执行过程中应该运行的各个步骤：

1. 从 source 开始，获取账户编号的流。
2. 管道的下一个阶段将与 AccountRepository 发生交互，并且将编号与 Account 对象进行映射。
3. 得到一个广播结合点，它将账户的输入流分发到两个独立流中。每个输入的账户会被同时分发给两个下游消费者。
4. 现在有了两个独立的流，在这里为每个账户应用转换来提取交易。一个取得

银行交易，另一个取得结算交易，其结果来自于证券交易的现金结算。通过广播，模型可以并行地获取两个交易流，也可以最优地利用计算资源。

5. 在完成交易的收集之后，通过一个合并（Merge）结合点来做合并。两个流重新组合成一个交易的流。通常情况下，合并不会在输出中保留输入的顺序。但某些合并操作如 MergePreferred 或 FlexMerge 可以帮助我们精准地控制顺序。
6. 应用另一个转换来校验所有交易。
7. 再次广播。一个流记录了所有交易日志作为审计线索。通过一个 sink 对其建模，它将取到所有没有经过任何改动的审计线索。而另外一个 sink 将对每个账户计算交易净值。

下面是通过使用图形生成 DSL 来构造这个管道图形的代码：

```
val graph = RunnableGraph.fromGraph(
  GraphDSL.create(netTxnSink) { implicit b => ns =>
    import GraphDSL.Implicits._

    val accountBroadcast = b.add(Broadcast[Account](2))
    val txnBroadcast = b.add(Broadcast[Transaction](2))
    val merge = b.add(Merge[Transaction](2))

    val accounts = Flow[String].map(queryAccount(_, AccountRepository))
    val bankingTxns = Flow[Account].mapConcat(getBankingTransactions)
    val settlementTxns = Flow[Account].mapConcat(getSettlementTransactions
    val validation = Flow[Transaction].map(validate)

    accountNos ~> accounts
               ~> accountBroadcast
               ~> bankingTxns
               ~> merge
               ~> validation
               ~> txnBroadcast ~> ns
    accountBroadcast ~> settlementTxns ~> merge
    txnBroadcast ~> audit
    ClosedShape
})
```

构造一个 RunnableGraph，这个图后续可以与实现器一起运行。

建立 Broadcast 结合点，参数 2 表示它是一个双路广播。

建立 Merge 结合点，参数 2 表示它是一个双路合并。

增加流。

连接并形成一个图。

前面的代码看起来还是比较直观的。再看看构造图形的 DSL，它和在白板上画图的方式几乎完全一样。构造完成之后，就可以通过 graph.run() 来运行整个图形。最后需要再看一下两个 sink，在这里将计算每个账户的净交易值与审计线索。netTxnSink 计算每个账户的交易净值，而 audit 记录日志。在当前的例子中，audit 在控制台进行打印，而在现实的场景中会有更复杂的记录器来做这个工作：

```
val netTxnSink: Sink[Transaction, Future[Map[String, Transaction]]] =
  Sink.fold[Map[String, Transaction], Transaction]
   (Map.empty[String, Transaction]) { (acc, t) =>
  acc |+| Map(t.accountNo -> t)
}

val audit: Sink[Transaction, Future[Unit]] = Sink.foreach(println)
```

netTxnSink 是一个 Sink，它从上游图节点获取 Transaction 并对它们执行 fold，生成一个包含每个账户净交易值的 Map。事实上，Transaction 是一个 monoid，Map 是一个 monoid，而它的值也是一个 monoid。剩余的实现部分是不言自明的。

所有这些代码都可以在本书代码库中找到。

6.4.3 后端压力处理

如前文所说，在流处理模型中有一个很重要的方面需要关注的就是后端压力处理。通常遇到的情况就是消费者（Sink）不能跟上生产者（Source）的生成速度。如果没有恰当的处理，这很有可能导致消费者要么丢数据，要么就崩溃并报 OutOfMemoryError 错误。

Akka Streams 以及其他一些实现提供了相关策略来应对后端的压力。[1] 当数据流从 Source 流向 Sink 时，请求则向相反方向流动。也就是 Sink 在不断刷新 Source 在当前时刻它所能承受的节流阀。这个后端压力信息持续地从消费者（Sink）流向生产者（Source），因此生产者就可以不断调整通过节流阀的流量。在文档中可以了解关于如何为模型调节后端压力处理的相关配置的具体细节。

本章解释了基于流管道的基本工作模型。不过我们还没有讨论在多个协同网络节点或无限输入流的情况下这个模型是如何工作的。在前面章节的例子中，所有的输入都是有限的固定序列，而且也还没看到后端压力处理是如何工作以及如何监控它确实是 sink 根据需求生成给 source 的。第 7 章提供了一个完整的实现，覆盖了将流处理管道设计为领域模型的组成部分所涉及的方方面面。这个例子组合了 actor 模型与基于流的模型。将 actor 输入到流处理管道中，根据领域规则对数据执行转换，并最终在实现领域行为的 sink 中终结。但在这里还是先列出我们目前讨论的流模型的几个重要结论：

- *巨大而且无限的数据*：在大数据时代，我们设计的服务通常需要面对无限或至少是巨大的数据流。不能指望把所有数据存储下来之后再去处理它。在这种场景下流模式是一个可行的选择。

1 参见响应式流（www.reactive-streams.org）。

- *异步与非阻塞*：就像基于 future 的计算，流提供了异步非阻塞的 API。这确保了模型对用户的响应性，以及计算资源的最佳优化。
- *灵活的 API*：流模型是基于位置透明的抽象，如 `Source` 和 `Sink`。我们可以配置服务并将它们指向管道的 `Source`。
- *后端压力处理*：在流模型中，数据流从 `Source` 流向 `Sink`，而请求则反向流动。消费者（`Sink`）发送请求给生产者（`Source`），生产者将根据该请求调整节流阀。`Sink` 持续地保持 `Source` 更新它的请求，这样 `Source` 就可以控制节流阀下游的流量。这就控制了后端压力，同时防止 `Sink` 被数据压垮。

在下一节中，将简要地学习 actor 模型，这也是本书中最后一个并发模型。

6.5 actor模型

actor 提供了一个基于实体间消息传递的计算模型。任何实体都可以发送消息给 actor，它将信息收进它的收件箱中，在收到消息时，actor 可以采取以下措施：

- 从它收件箱的头部处理一个消息
- 创建新的 actor
- 更新它的状态
- 发送消息给其他 actor

actor 通常被实现为一个极度轻量的构造。一般来说，可以在标准配置的笔记本电脑中创建上百万个 actor，而且它们可以并行处理消息。actor 首先在 Erlang 编程语言（www.erlang.org）中得到普及，这个模型基本上被它当成基本的并发产物在使用。受到 Erlang 的启发，Akka 将 actor 模式带入了 JVM。尽管在设计上它们有所不同，Akka 的 actor 与在 Erlang/OTP 平台上所看到的还是基本相同的。

在本节中，将主要从领域建模的视角来观察 Akka 的 actor。本书不会在这里囊括 Akka 提供的所有特性，如果需要，可以参考 Raymond Roestenburg 的 *Akka in Action*（Manning, 2015 年）。本章重点讲述 actor 模型的能力，这对于设计领域模型来说非常有用，而且还会指明需要规避的弱点。让我们先看下 Akka 中的 actor 所提供的一些主要特性：

- *基于异步消息*：在 Akka 中，actor 是异步的、消息驱动的并且是非阻塞的，这完全符合响应式计算的定义。发送给 actor 的消息通常是类型化的不可变实体，它会被添加到 actor 的收件箱中。需要定义一个能够处理消息类型的接受循环，这样 actor 就可以做一些相关处理。
- *单线程执行*：actor 一次处理一个消息，在控制上它是单线程的。不需要在 actor 的消息循环内实现任何并发。可以在 actor 里拥有可变状态，而且非常完美。保护可变状态是 actor 模型中非常常见的场景之一。

- 监管：Akka 中的 actor 实现了监管层级。可以在 actor 之间定义父子关系，在这里父 actor 负责管理失败的子 actor。当一个 actor 发生失败并伴有一个异常时，父 actor 决定是否重启子 actor，这依赖于 actor 系统声明的监管策略。这样做的好处就是它使失败处理由 actor 的结构负责，而不是由某些指定的 actor 负责。这也是 Akka 从 Erlang/OTP 平台借鉴来的思路。
- 位置透明：发送消息给 Akka 的 actor 时，不需要知道接收 actor 是否与发送者部署在一起。使用相同的语法规则发送消息，消息将被分发给 actor，哪怕它在另外一个网络节点中。这是发送者和接收者之间的彻底解耦。
- 有限状态机：可以使用 `become` 方法动态改变 Akka 中 actor 的行为。使用计算的 actor 模型非常有助于建立有限状态机模型。

6.5.1 领域模型与 actor

我们想要建立响应式领域模型，而且也看到了一些可选的方法。actor 提供了另外一种并发与异步的模型，用于构建可扩展的响应式领域模型。但这个模型在原理和架构上与我们之前所讨论的所有其他模型都有一些区别。这源于一个事实，就是 actor 最初被认为是一种设计需要大规模并发和可扩展性的系统的手段。实际上它首先在动态类型编程语言中得到普及，这也使我们相信，actor 从来没想过作为一种架构要应用于模型的简单代数推理。在静态类型编程语言中实现 actor 时，这是一个缺点吗？当我们更深入地挖掘 actor 模型在设计响应式领域模型的适用性时，就会找到相关答案。

Actor 与类型安全

Akka actor 在行为上不是类型安全的。`receive` 是定义 actor 行为的基本方法，如果阅读一下它的结构，就会发现它被定义为 `PartialFunction[Any, Unit]`，这意味着对于 actor 最终会做什么没有任何静态的保证。而这样的设计有以下两个原因。

- *FSM*：Akka 的 actor 可以用 `become` 方法建模有限状态机（FSM），这意味着 actor 可以根据当前状态以及它所接收的消息转变它的行为。这本质上就需要一个动态行为框架。牺牲了类型安全，但获得了更多的能力。当设计 API 时要时刻记得这个，并试着用 Akka actor 建立它们的响应性。这并不邪恶，实际上有很多用例就是使用这种额外的能力在一个简洁的特定领域语言中实现状态机。
- 分布：Akka 的 actor 可以透明地分布在各个节点。这就要求 actor 能够动态地序列化，也因此很难对它们引入任何静态类型保证。

相对的，Scalaz 也有一个 actor 实现，它比 Akka 的 actor 提供了更多的类型安全。[1]`Actor[A]` 只能接受类型 `A` 的消息，这里我们通常将 `A` 定义成一个有多个子类型的封闭特质。这个妥协的后果是 Scalaz 的 actor 既不能被用来建模 FSM，也不能透明地跨节点部署。

Actor 与可组合性

类型 `PartialFunction[Any, Unit]` 的行为只针对副作用。[2]这也并不奇怪，因为缺乏引用透明，所以 Akka 的 actor 不能组合。因此不能在模型中推导 Akka 的 actor。在能用可组合、引用透明的 API 管理的地方，就不要使用 actor 了。

Actor 与响应式 API 设计

缺乏类型安全与可组合性使得 actor 在设计终端用户 API 方面不是一个很好的选择，特别是在使用 Scala 这种静态类型编程语言时。但对于一些特定的使用场景，可以拥有一些终端用户 API 基于 actor 的实现。当需要在可变状态上强制互斥时，这一点尤其正确，而且它工作起来像一个链条，因为每个 actor 遵守单线程的执行模型。这也是最推荐采用 actor 的使用场景。

来看下面的例子，在这里需要从一个异步消息流中追踪最大值的 debit 交易。可以将这个追踪器建模为一个 actor，它将最大值的交易保存为一个可变状态。对每一个接收到的消息进行检查，如果需要就改变最大值。

```
class MaxTransactionTracker extends Actor {
  var max: Transaction = _

  def receive = {
    case t @ Transaction(_, _, transType, amount, _) =>
      if (transType == Debit && amount > max.amount) max = t
    case MaxTransactionSoFar => { sender ! max }
  }
}
```

现在可以定义一个终端用户 API，这个 API 的实现基于前面的 actor。通过单线程执行，actor 确保可变状态（最大值的交易）得到保护并且只能通过消息发布给发送者。

在某些情况下，会发现在通过 actor 和 future 实现的某些行为之间存在强并行。比如下面这个 actor 的例子，它提取用户投资组合中的指定证券：

1 更多关于Scalaz的actor信息，参见https://github.com/scalaz/scalaz。

2 `PartialFunction[Any, Unit]`的意思是它可以获取一个`Any`类型的参数并生成一个`Unit`。这种函数不能组合，因为类型`Unit`不允许组合。

```
sealed trait Instrument
case class Equity(..) extends Instrument
case class Currency(..) extends Instrument

class PortfolioReporter extends Actor {
  def receive = {
    case GetPortfolio(acc, ins) => //..
    //..
  }
}
```

现在要为一个指定客户提取普通股与货币投资组合，并将它们作为一个整体组合进行报告。下面是我们的首次尝试（也是个失败的尝试），它使用 actor 的 ask 方法，返回一个 Future：

```
val eq = Equity(..)
val ccy = Currency(..)
val pEquity = prActor ? GetPortfolio(acc, eq)
val pCurr   = prActor ? GetPortfolio(acc, ccy)
for {
  e <- pEquity
  c <- pCurr
} yield merge(e.asInstanceOf[Portfolio], c.asInstanceOf[Portfolio])
```

结合现在所掌握的 actor 模型的知识，这是一个可接受的响应式 API 模型吗？actor 以单线程的方式执行，尽管 pEquity 和 pCurr 都是 Future，这两个消息还是被串行地处理。这里命中了同一个 actor，因此在执行上也没有任何并发性。考虑 actor 的执行模式，解决方案就是创建独立的 actor 以获取执行中所期望的并行水平。

```
val prActor1 = system.actorOf(Props[PortfolioReporter])
val prActor2 = system.actorOf(Props[PortfolioReporter])[1]
//.. same as earlier
```

现在考虑用基于 Future 的 API 实现相同的例子：

```
trait PortfolioReporter {
  def getPortfolio(a: Account, i: Instrument): Future[Portfolio]
}

val pr: PortfolioReporter = //..

val eq = Equity(..)
val ccy = Currency(..)
```

1 这里做了简化，实际情况下可以使用RoundRobinPool(2).props(Props[PorfolionReporter])作为更符合语言习惯的实现。更多细节参见Akka文档。

```
val prEquity = pr.getPortfolio(acc, eq)
val prCurr = pr.getPortfolio(acc, ccy)

for {
  e <- prEquity
  c <- prCurr
} yield merge(e, c)
```

这是不是更加简单？不再需要定义 actor，也避免了它们所带来的样板文件。API 更加简洁，同时有更好的组合性。所以通常的建议是使用 Future 而不是 actor。Scala 中的 Future 具备了组合能力，而且为声明式编程提供了大量的组合器。总之，Future 可以组合，actor 不可以。

Actor 与边界上下文

在本章的前半部分，我们学习了如何在便捷上下文间使用异步消息进行通信。如果使用像 Akka 这样的 actor 框架，完全可以借助 actor 的力量来完成这事。actor 的基础架构包含了收件箱和消息传递，因此不再需要管理自己的队列。对于通信的协议，可以设计不可变消息并用 actor 来传递它们。Akka 通过远程 actor 提供了位置透明，通过集群提供了冗余。这些都有助于在边界上下文之间建立可信赖的消息传递系统。

Actor 与弹性模型

一个成熟的 actor 模型是一个粗粒度的抽象。actor 会有一些负载，特别是在同一个框架中包含如监管、远程调用、分布式以及集群等特性时。在选择 actor 在模型中作为处理并发的基本手段之前，必须意识到这些后果。

但是，Akka 所提供的这些附加特性组成了模型架构中某些重要部分的基石。当有一个复杂模型时，会希望在面对失败时它能保持足够的弹性。就像在第 1 章中所讨论的，就算某些应用组件失效了，也不希望用户体验被降级。只有实现了“随便它崩溃”的哲学，才有可能达成期望。我们可以在代码中明确处理一部分失败。但硬件或网络层的失败就超出了应用的控制范围。唯一的办法就是在架构中将失败作为一个独立的关注点来对待，并由专门的组件来处理它们。

遵循 Erlang/OTP 的原则，Akka 允许监管被实现为模型中单独的架构。可以定义 actor 的监管层级，它将专注于处理子 actor 中可能产生的异常。根据异常类型，可以为子 actor 指定要采取的措施，如继续执行、终止或重启。因此，如果一个 actor，假设叫 S，被声明为一系列 actor[c1, c2, c3] 的监管（父），那么任何子 actor 处理过程中所产生的异常都会被通知到 S。S 将根据失败的类型采取相应的处理措施。这个方法的美妙之处就在于 c1、c2 或 c3 的行为不再与异常处理的代码纠缠在一起。

Actor 与函数式领域模型

正如大家所看到的，actor 不能组合、有很多副作用，并且在模型的工作方式上有很多的不确定性。那么当设计函数式领域模型时，actor 在并发方面有没有起到什么好的作用？

本章之前讲过，哪怕在函数式领域模型中，actor 也扮演了一个非常重要的角色，具体体现在以下 3 个方面：

- *保护共享可变状态*：首先是作为共享可变状态的容器，actor 通过单线程执行确保了互斥，这已经深深扎根于我们的实现中。边界 API 可以是函数式的，而且不会暴露 actor 操作的任何可变状态。这种模式在实现层面守护着变化，因此不会影响到 API 的纯洁性。
- *弹性*：将 actor 用作粗粒度的抽象，以便对弹性与冗余架构进行建模。在最顶层，我们用监管保护其他顶级服务 actor。这些服务 actor 仅仅是调度程序，它将所有领域行为委托给覆盖下的纯函数实现。所以，函数式领域模型是纯洁的，同时还有一个 actor 的 façade 来提供模型的弹性。
- *与 Akka Streams 一起玩耍*：使用 Akka Streams 提供的类型模型而不是使用 actor 来设计 API，将会更加安全，而且可以具备组合性。接着通过 `ActorMaterializer` 用 actor 实现那些 API。我们享受了轻量级可扩展的 actor 所带来的好处，同时还为最终用户发布了类型化的 API。6.4 节讨论了流的使用案例，在第 7 章会看到一个更大的用例。

说到这里，很多开发人员还在用更普遍的方式使用 actor 模型。在领域驱动设计中，一个普遍应用就是将聚合根建模为一个 actor，然后用 Akka Persistence 来实现领域实体的持久化（http://doc.akka.io/docs/akka/snapshot/scala/persistence.html）。这个技术可用于事件溯源以及基于命令查询责任分离（Command Query Responsibility Segregation——CQRS）的架构（将在第 8 章中讨论）。这个方式强制将聚合根建模为 actor，而不是简单代数数据类型，到目前为止我们一直鼓励这样做。也正如其他技术一样，它也有自己的优缺点。第 8 章将详细描述这种模式如何与函数式建模的精神相互作用，以及一个简单的变化是如何保留所有看上去将要失去的优点的。

6.6 总结

本章讲述了如何建立响应式领域模型，这也开始了本书的第二部分。我们学习了 Scala 中许多用于建模的选项。本章的主要结论如下：

- *什么是响应式领域模型？*学习了响应式模型应该具备的特征，以及如何在终端用户 API 中实现它们。还学习了 JVM 上的选项以及它们的利弊。

- *future* 与 *promise*：学习了如何使用 future 和 promise 设计响应式非阻塞 API。我们看到了个人银行领域中的一些场景，还实现了一些模式，使模型具有良好的响应性。
- 异步消息传递：对于那些在时间和空间上解耦的系统来说，这是它们之间进行通信的最常用的一种技术。而且领域模型中多个边界上下文提供了相关配置来使用这种模式。
- 流模型：流提供了类型抽象，用于建模领域行为管道。学习了领域中一些可以被映射到此类实现的使用场景。为此目的将使用 Akka Streams。
- *actor* 模型：在 Scala 中，这可能是 Akka 引入的最常用的并发产物。尽管 actor 可以使领域模型具备可扩展性，但它们也不是完全没有问题。actor 都是副作用，它并不总是与函数式编程的精神和睦相处。但在保护共享状态以及领域模型的容错方面，它们依然提供了一些有价值的特性。

7 响应式流建模

本章包括

- 响应式流基本设计原理回顾
- 使用响应式流作为主干的完整用例实现
- 将其他技术，例如 actor 模型与响应式流组合在一起的方法
- 详细讨论通过流来进行响应式建模的基础知识

在前面的章节中，学习了响应式建模的基本原理以及使模型能够保持响应能力的技术。在这一章节中，将会深入学习响应式流模型与一个使用 Akka 流的用例实现。我们将从业务需求开始，将它们映射到流处理的各个阶段，并逐步实现各个组件。用例来自于个人银行领域，涉及跨边界上下文的异步边界处理，这为讨论诸如响应式套接字管理和后端压力处理提供了完美设定。我们所讨论的领域行为还涉及数据转换，它也适合于使用流模型来实现。本章包含了使模型具有响应性的各个方面，并展示了实现将如何处理它们。

图 7.1 展示了本章内容的纲要。这个指南将帮助大家在浏览本章时选择感兴趣的主题。

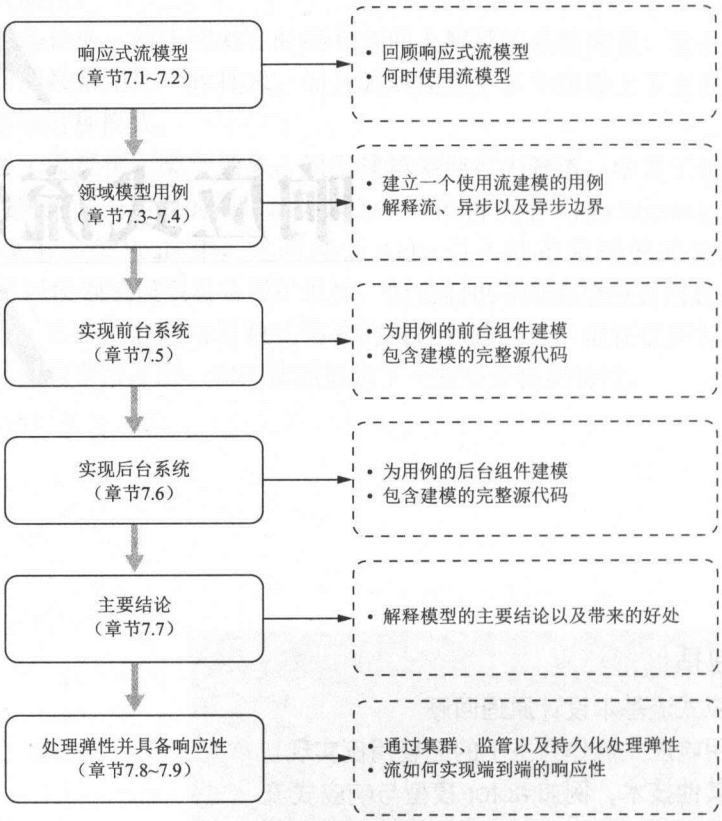


图 7.1 本章学习过程。

在本章结束时，应该能够知道如何利用合适的抽象使领域模型对用户保持响应性以及对失效情况保持弹性。

7.1 响应式流模型

从迄今为止的学习中，不难想象领域模型是一个图 G，它的节点是各种组件，例如实体、聚合、值对象以及服务，这些组件通过领域行为交互的边缘建模被关联在一起。这里我们说的是一个具有特定领域和数据模型的边界上下文，而且边缘建模的交互可以是同步或异步的。

就更高层次而言，也可以将多个边界上下文想象成另一个图形 H，其中每个上下文形成一个节点，并且连接两个上下文的边界对它们如何交互进行建模。G 和 H

之间的主要区别在于，在H中连接多个上下文的边界几乎只会形成系统的异步边界。[1]

在第6章中，我们讨论了消息传递可以作为处理跨模型异步边界通信的一种方法。还看到基于明确消息传递使用actor模型设计API的缺点。actor提供了一个无类型的交互模型，这导致领域模型无法推导。但是因为actor是轻量级的，所以它们很容易扩展，这与线程不同，我们可以在标准笔记本电脑上运行上百万个actor，传递实体间领域模型的交互事件。

响应式流模型提供了一个两全其美的方法：使用构成声明式DSL的类型API，它具体化为actor模型的实现。可以对同步和异步边界统一建模，以一个原则的方式处理失败，但仍获得使用组合类型API进行推理的所有优势。作为响应式模型，流将处理后端压力，确保从生产者流向消费者的数据与反向流动的需求之间保持最佳的平衡。

在接下来的几节中，将详细了解响应式流以及如何将它们与其他并发模型相结合。还将看到，流模型作为建立响应式架构的完整解决方案，是如何解决弹性、灵活性和扩展性的问题的。

7.2 何时使用流模型

一旦了解了什么是流模型，接下来的问题便是何时去使用它。毕竟流提供了一个具有一定成本的抽象。显然，我们更愿意用适合用例的最廉价模型。和其他抽象一样，流的使用也不是完全免费的。

我们可以将领域模型的动态行为可视化为处理单元之间的交互图。在流计算的词汇表中，这些处理单元通常被称为运算符，而交互被称为管道。可以将一个业务用例可视化为流过管道的数据流并通过一组运算符进行转化。在下一节中将更多地讨论这样的用例及其基于流的建模。

当我们有一个用例，它可以被建模成一系列转换并形成流程图，那就可以考虑用响应式流作为潜在的建模抽象。这个模型的本质是它使流成为一个头等类抽象，从而使运算符从数据何时到达何时离开的关注中解耦出来。数据可以以异步或同步的方式处理，我们的运算符完全忽略这一事实。这使模型更加模块化，因为我们已经将运算符与数据流解耦，并且两者都可以在模型中独立进化和扩展。作为示例，如果查看Akka Streams的架构，`Flow`会对流处理的步骤进行建模，并提供单独的函数`map`和`mapAsync`来描述运算符之间的数据流类型（同步或异步）。`map`和`mapAsync`是不同的运算符，它们可以对`Flow`进行操作，从而使整个架构更加模块化，并将`Flow`从被操作中解耦出来。

1 关于异步边界的定义，请参见6.3节。

7.3 领域用例

让我们从个人银行领域建立一个用例，使用响应式流作为主要实现技术来建模，考虑银行前台与其他外部系统间的交互，以及后台处理所有合并并生成记录系统。

以下是我们将要建模的工作流：

- *聚合交易*：前台将各个外部系统交易汇总后，得到一个合并的交易列表。假设此聚合将会与前台作为一个以逗号分隔的文件出现，它可以出现在任何地方（在消息队列或另一个持久性存储中）。但为了简单实现，我们假设它是一个文本文件。
- *发送到后台*：前台需要将交易数据发送到后台做进一步处理和结算。
- *创建领域模型元素*：后台从文件中接收交易信息，并创建领域对象，它可以通过适当的领域行为与其他的模型部分进行交互。
- *净交易*：后台需要每个帐户的净交易。交易可以是 debit 交易或 credit 交易。对于 debit 交易，资金将从帐户中扣除，而对于 credit 交易，资金将被增加到账户中。
- *持久化和通知*：净交易需要保存到存储中。因为前台和后台之间的这种交互会持续很长时间，所以后台需要周期性地发布所有净交易的状态。在现实中，这种发布可能表现为仪表板的形式，但这里为了简单起见，可以考虑只做打印。

这个用例在交互系统之间建立了一个共同的行为模型。我们需要处理前台和后台之间的异步边界，考虑的重点是这种交互的波形可能相当尖锐。在营业高峰时间，前台将被加载大量的交易，到后台的数据流量也会很高。重要的是后端系统能够处理交互请求的流量，这是响应式流处理的主要问题之一：后端压力。让我们跳到模型看看如何解决这个问题。

7.4 基于流的领域交互

到目前为止，我们省略了在持续净交易之前前台和后台都需要做的一些转换。在详细介绍它们的实现之前，让我们来看一个基于流的交互图，这可能比一个详细的描述更直观。图 7.2 描述了模型的整体架构。

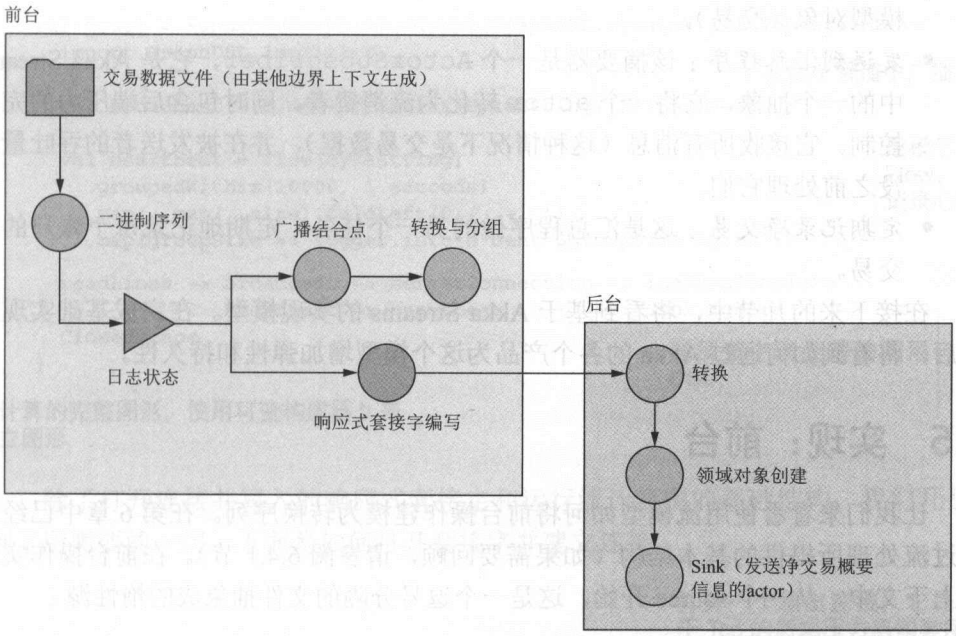


图 7.2 使用响应式流建模净交易用例。后台组件与前台有一个异步边界。前台会接收一个文件中的交易数据，然后将其传输到后台做处理、计算净值并持久化。

让我们深入这个架构，并使用 Akka Streams 来实现用例（http://doc.akka.io/docs/akka/2.4.4/scala/stream/index.html）。[1] 在完成实现之后，将感受到它带来的价值，以及它通过原生 actor 和消息传递来改善编程模型的方式。在着眼于实现之前，先看一下发生在前台与后台系统处理中的转换步骤。前台和后台是两个独立的边界上下文，它们之间有严格的异步边界。

- 序列化到二进制：所有的交易数据需要被传输到后台系统。为了高效地做转换，将其序列化为字节流。这是发生在输入数据上的第一次转换。为简单起见，会假设在转换成逗号分隔的文件之前，交易数据就已经得到过验证。
- 分组和日志：前台处理开始记录所有传输的数据。为了优化，将分组日志记录，分批进行。
- 传输：前台处理需要写入响应式套接字以传输到后台。响应式套接字有一个内置功能来处理后端压力。注意，接收方（后台）也绑定到相同的套接字以便读取数据。
- 解析和拆分：后台系统接收二进制数据并将其转换回文本。然后在逗号分隔数据的基础上再次分离数据。
- 创建交易：转换的下一个步骤将获取每个独立的记录，并将它们转换为领域

1 Akka Streams是基于响应式流（Reactive Streams）规范的（http://www.reactive-streams.org）。

模型对象（交易）。

- *发送到汇总程序*：该摘要器是一个 ActorSubscriber，它是 Akka Streams 中的一个抽象，它将一个 actor 转化为流消费者，同时包含后端压力的完全控制。它接收所有消息（这种情况下是交易数据），并在被发送者的吞吐量淹没之前处理它们。
- *定期记录净交易*：这是汇总程序做的另一个任务，定期地记录每个帐户的净交易。

在接下来的几节中，将看到基于 Akka Streams 的实现模型。在完成基础实现工作后，将看到如何通过 Akka 的各个产品为这个模型增加弹性和持久性。

7.5 实现：前台

让我们来看看使用流模型如何将前台操作建模为转换序列。在第 6 章中已经讨论过流处理所提供的基本结构（如果需要回顾，请参阅 6.4.1 节）。在前台操作实现的上下文中，从一个 source 开始，这是一个逗号分隔的文件抽象成的惰性源：

```
val path = ..
val getLines = () => scala.io.Source.fromFile(path).getLines()
val readLines = Source.fromIterator(getLines).filter(isValid)
  .map( l => ByteString(l + "\n"))
```

文件路径

将文件转换成惰性抽象

转换成响应式源，它是转换链条中的一个运算符

这里最终会得到一个 Source，它将产生一个传输到后台系统的字节流。isValid 是一个校验函数，它被用来校验逗号分隔文件中的记录并过滤掉无效的实体。后面会看到如何通过插入在流处理中处理异常的策略来处理无效记录。

现在有了源，需要在多个管道里进行广播。一个需要写入套接字以便发送到后台，而另外一个则记录日志用于心跳检查。对于心跳，我们定期分组记录日志，因此需要构造一个 Flow 来照顾这个转换。同时这个 Flow 将终结在一个 Sink 里，这个可以暂时先忽略。现在需要在一个图中编排所有这些序列，并通过一个响应式套接字连接来运行它。

```
val logWhenComplete = Sink.onComplete(r =>
  logger.info("Transfer complete: " + r))
```

记录传输过程完成状态的 Sink

```
val graph = RunnableGraph.fromGraph(GraphDSL.create()) { implicit b =>
  import GraphDSL.Implicits._

  val broadcast = b.add(Broadcast[ByteString](2))

  val heartbeat = Flow[ByteString]
    .groupedWithin(10000, 1.seconds)
    .map(_.map(_.size).foldLeft(0)(_ + _))
    .map(groupSize => logger.info(s"Sent $groupSize bytes"))

  readLines ~> broadcast ~> serverConnection ~> logWhenComplete
               broadcast ~> heartbeat        ~> Sink.ignore
  ClosedShape
}
```

流计算的完整图形，使用可变构建器 b 来建立图形

在两个管道中广播

建模了 Sink 的 Flow，在这里记录心跳

建模整个流处理的图形 DSL

除了打开连接并写入的响应式套接字和运行流计算图的基础结构，我们几乎得到了所需要的一切。下面是如何打开套接字并建立连接：

```
implicit val system = ActorSystem("front_office")
val serverConnection = Tcp().outgoingConnection("localhost", 9982)
```

打开一个输出连接，它是一个基于 Tcp 的后端压力感知套接字

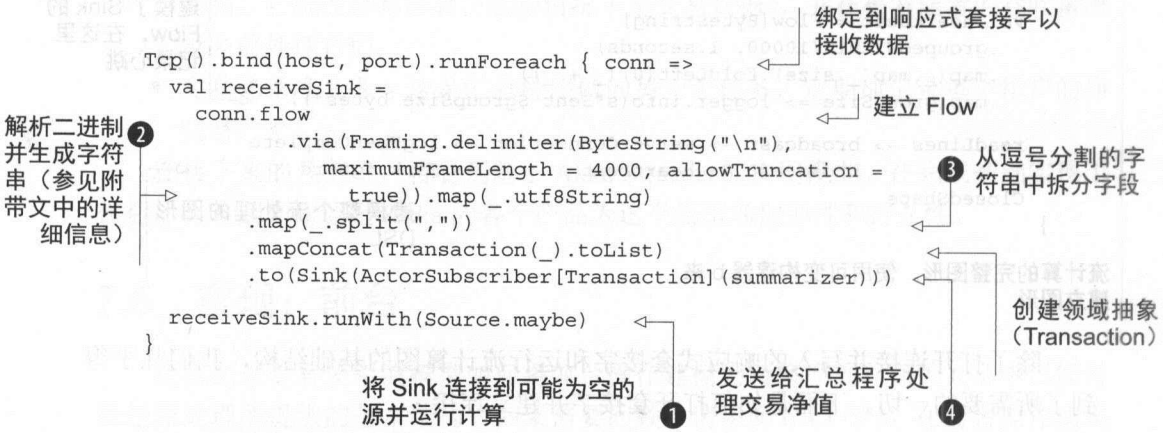
这就是如何使用一个 ActorMaterializer 运行图形。正如在第 6 章中看到的，ActorMaterializer 通过构建底层的 actor 来运行图形。注意流计算框架如何从用户 API 抽象 actor 模型。如第 6 章中所述，当 actor 被部署为实现媒介时，可以发挥最大的作用。完整版本的前台可运行代码可以在本书的代码库中找到。

7.6 实现：后台

本示例演示了如何通过各种技术的结合，使一个解决方案在边界上下文内部或跨上下文方面，都有良好的响应性与模块化。我们已经看到使用响应式流作为实现骨干的前台实现（尽管只是一个简单的用例）。对于给定的一个用例，如果它符合基于流的实现的要求（如 7.2 节中所讨论的），那就必须争取使用它。正如被 Akka Streams 所实现的流，它是强类型的（不同于 actor 模型），但是可以获得 actor 提供的可扩展性的好处。这是否意味着我们不会在任何模型元素中明确地使用 actor？正如在第 6 章中所看到的，actor 仍然可以作为某些用例的显式编程模型。所以，不要陷入特定的偏见。根据情况需要合理地使用所有工具。在当前这个后台实现的用例中，将看到如何使用 actor 来解决一个对其他技术来说非常繁杂的问题。

7.4 节中列出了我们系统设计的任务（实现级别）。最后 4 个要点涵盖了后台实现将要处理的项目。让我们将这些步骤看作为流处理管道的一部分。

以下代码显示了开始从前台接收数据之后进行转换的核心步骤：




```
Tcp().bind(host, port).runForeach { conn =>
  val receiveSink =
    conn.flow
      .via(Framing.delimiter(ByteString("\n"),
        maximumFrameLength = 4000, allowTruncation =
          true)).map(_.utf8String)
      .map(_.split(","))
      .mapConcat(Transaction(_).toList)
      .to(Sink(ActorSubscriber[Transaction](summarizer)))

  receiveSink.runWith(Source.maybe)
}
```

正如之前所看到的，Akka Streams 提供的 API 是声明式的，它很容易就可以表达设计者的意图。即使不知道实现的细节，依然可以很直观地了解它试图实现的内容。为一个很复杂的领域建模时，这样的 API 对我们来说就非常有帮助了。

我们有一个空的 `Source`❶，但我们不需要，因为我们是从套接字中读取字节的。有趣的部分是 `Flow`，它从字节的解析开始 ❷。这里不会淘汰太多关于解析的细节，因为 `Framing` 类提供的好助手已经很好地进行了抽象。如果有兴趣，可以阅读 Akka Streams 的文档和代码。

在得到字符串格式的记录后，需要跨逗号进行拆分 ❸。这将为我们提供用于构建 `Transaction` 对象的单独字段。

最有趣的事情发生在开始获得 `Transaction` 对象之后，需要将每个交易传递到可以进行汇总的地方。注意，它需要在某处存储汇总记录（`Balance` 为净交易），这意味着它需要管理一个状态（可能是可变的）。它可以将状态存储在内存或数据库中，但重点是操作这个状态时要确保互斥。如果还记得从第 6 章开始讨论的响应式模式，这就是 actor 一个的理想使用场景。在 ❹ 中，`summarizer` 是一个 actor，但不是一个普通的 Akka actor，这是一个响应式后端压力感知的消费者抽象。[1] 让我们在清单 7.1 中看看这个 actor 的 `receive` 循环。

1 可以在Akka Streams的文档中查阅`ActorSubscriber`的有关内容。

清单 7.1 使用单子组合器生成有效的账户编号

将净交易存储为一个可变 Map。actor 管理对可变状态的单线程访问。

我们的 actor 是一个 ActorSubscriber，这意味着它是一个响应式后端压力感知的 actor 抽象。

```
class Summarizer extends Actor with ActorSubscriber with Logging {
  private val balance = collection.mutable.Map.empty[String, Balance]

  def receive = {
    case OnNext(data: Transaction) =>
      balance.get(data.accountNo).fold {
        balance += ((data.accountNo, Balance(data.amount,
          data.debitCredit)))
      } { b =>
        balance += ((data.accountNo, b |+| Balance(data.amount,
          data.debitCredit)))
      }
    //..
  }
}
```

接收循环接收 Transaction 对象。

通过一个 monoid 计算净交易。请查阅代码库，了解如何在 monoid 内抽象处理 debit 和 credit 交易的逻辑。

所以现在使用流管道中一个 actor 来处理一个行为，而这个行为最适合由 actor 抽象来处理。这个实现的主要结论是，应该总是选择手头上最适合用于使用场景的抽象。在这个用例中，需要在后台中建模一个行为流，因此要选择提供声明式图形 API 的流管道。在整个流中，针对流管道中的不同阶段需要特定的处理能力。同时，因为需要管理可变状态，所以还使用了 actor。

Summarizer actor 管理了净交易的状态。在示例中，将维护一个可变的 Map，它计算交易净值并为每个帐户保存一个 Balance。或者，也可以更新数据库来达到同样目的。我们需要解决的最后一个问题就是要记录（也可能显示在仪表板中）净值处理的进度。这个事情完全与流处理序列的时间线解耦，它是一个彻底的异步过程。可以再次使用 Summarizer actor 接收一个特殊消息来完成此项作业，然后通过使用某种调度服务来调度此消息分发。下面是执行日志记录 Summarizer actor 的部分内容：

```
class Summarizer extends Actor with ActorSubscriber with Logging {
  //.. as before
  def receive = {
    case OnNext(data: Transaction) => //.. as before
    case LogSummaryBalance => logger.info("Balance so far: " + balance)
  }
}
```

可以使用后台处理路径来安排此消息的调用。清单 7.2 显示了这方面的内容，并展示了建立与后台交易转换管道相绑定的响应式套接字的初始步骤。

清单 7.2 交易处理流管道

```
import akka.actor.{ActorSystem, Props}
import akka.stream.ActorMaterializer
import akka.stream.actor.ActorSubscriber
import akka.stream.io.Framing
import akka.stream.scaladsl.{Tcp, Source, Sink}
import akka.util.ByteString

import scala.concurrent.duration._

class TransactionProcessor(host: String, port: Int)
  (implicit val system: ActorSystem) extends Logging {

  def run(): Unit = {
    implicit val mat = ActorMaterializer()

    val summarizer = system.actorOf(Props[Summarizer])

    logger.info(s"Receiver: binding to $host:$port")

    Tcp().bind(host, port).runForeach { conn =>
      val receiveSink =
        conn.flow
            .via(Framing.delimiter(ByteString("\n"),
                 maximumFrameLength = 4000,
                 allowTruncation = true)).map(_.utf8String)
            .map(_.split(","))
            .mapConcat(Transaction(_).toList)
            .to(Sink(ActorSubscriber[Transaction](summarizer)))
      receiveSink.runWith(Source.maybe)
    }

    import system.dispatcher
    system.scheduler.schedule(0.seconds, 1.second, summarizer,
     LogSummaryBalance)
  }
}
```

绑定响应式套接字。注意，使用与前台相同的主机与端口。

完整后台转换管道

使用 Akka 调度器调度汇总信息的记录。注意，它将每秒钟发送一个消息给 Summarizer actor。该 actor 将从它维护的状态中记录余额。

7.7 流模型的主要结论

流处理被证明是构建响应式系统的核心技术之一。正如前面讨论的，此模型提供了设计非阻塞 API 的所有特性，这对于使模型具有良好响应性是至关重要的。本章着重于实现类似今天所使用的、流计算模型同样适用的场景。我们看到两个边界上下文：前台和后台，它们在空间和时间上彼此解耦。然而，它们可以通过在响应式套接字之上使用流转换管道，被很好地集成到一起。这里总结了该模型的通用要点，以及基于响应式流规范的 Akka Streams 实现的特殊要求：

- *像流一样流动的数据建模*：对于许多使用场景（包括本章例子的更通用版本），可能会看到一次数据。这种处理模型建立了一个对数据进行处理的连续非阻塞转换管道。关于这种连续性如何处理或者流量控制如何被管理的细节完全取决于抽象。我们以声明的方式定义处理管道并发布要求，而基础设施负责其余部分。
- *声明式*：API 是声明式的。正如在示例中看到的，使用特定领域语言来实现流管道。这个词汇表与处理流管道的通用语言产生共鸣。例如，以 `Source` 开始，通过 `Flow` 转换，并在 `Sink` 中终止。
- *模块化*：由于 API 是声明式的，它们允许构建适合领域的单独构造。在构建图形和运行它之间存在清晰的分离。这使得模型组件可以被重用，并且模型整体上也是模块化的。
- *并发和并行*：每个处理阶段都通过 actor 实现。因为 actor 是轻量级抽象，可以对此实现的基础架构进行大量扩展。此外，处理阶段被有效地进行组合，以便提供最佳的吞吐量。比如，连续的 `map` 阶段被管道化，阶段之间的通信也通过使用它们维护的内部缓冲区而得到优化。`mapAsync` 被委派给一个 `Future`，因此这种昂贵操作的建模不会阻塞主线程的执行。当建模 `Broadcast` 时，再次通过 actor 模型并行执行各个阶段，从而确保最大的处理效率并提高吞吐量。但最棒的事情是，API 很好地抽象了所有这些实现细节，作为一个程序员，不会再为此感到困扰。
- *后端压力*：关于这个问题，我们已经讨论得足够多了。响应式流实现处理后端压力，因此消费者从来不会被数据淹没。诸如 Akka Streams 之类的响应式流实现内建了后端压力处理的响应式套接字，因此，根据数据流以及请求流，模型的拓扑可以适应不同的负载。这是伸缩性的本质，也是第 1 章中所讨论的响应式系统的一个重要特征。

7.8 使模型具有弹性

响应式领域模型的核心原则之一是弹性——模型对失败的恢复力。本节将展示如何使流处理管道对故障保持弹性。第 1 章详细说明了模型具有弹性的意义。除非在一些重大和不太可能的情况下，发生的任何故障都不应该导致整个系统发生瘫痪。失败总会发生，我们也不可能通过在领域模型中硬编码异常处理代码来防止每种类型的异常。

为了解决弹性问题，需要集中管理失败。有些东西有能力处理所有失败，或至少是那些想要被明确处理的失败。我们甚至可以指定在发生此类失败时要采取的动作。如果需要更复杂的错误处理，处理程序可以是分层级的，例如，错误可以跨监

管层传播，并且最终在指定级别处理。Erlang（www.erlang.org）几年前实现了这个策略，Akka 也将其引入了 JVM。

本节介绍了使模型具有弹性的 3 个方面，以及如何用响应式流实现它们：

- 提供有监管的故障管理。
- 处理节点或虚拟机的故障。
- 在面临失败时使数据持久化。

我们称之为*弹性的三驾马车*，它们使整个模型面对故障时具备运行时弹性，参见图 7.3。Akka 通过 actor 模型提供所有相关内容，我们可以在基于流的实现中使用它们。在后面的小节中将介绍详细信息。关于 Akka actor 中各个专题的更多信息，请参阅 Raymond Roestenburg 所著的 *Akka in Action*（Manning，2016 年）。

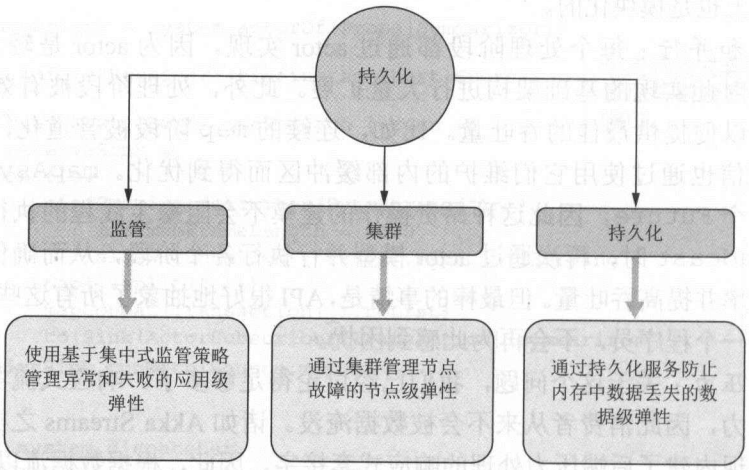


图 7.3 展示了 3 种形式的弹性。我们需要通过适当的、集中的、基于监管的故障管理来防止应用故障，通过 Akka 提供的集群来管理节点故障，并通过持久化 actor 来管理内存中数据的持久化。

7.8.1 使用 Akka Streams 监管

我们可以在基于流的模型实现中插入 Akka actor 模型所提供的监管策略。语义是完全相同的，并且当运行应用程序时，监管层次结构也将准备到位。我们在 6.5 节中简要讨论了监管策略。在本章前面介绍的基于流的模型中，可以指定具体的监管策略实现：

```
val decider: Supervision.Decider = {
  case _: FormatException => Supervision.Resume
  case _                  => Supervision.Stop
```

```
}
implicit val mat = ActorMaterializer(
  ActorMaterializerSettings(system).withSupervisionStrategy(decider)
```

在这个代码片段中，我们将模型的异常处理策略定义为双管齐下的方法。元素流动过程中（也许是从前台到后台），如果从 Source 读取的记录中得到任何 FormatException，接下来就只须从流中删除该元素并继续就可以了。这就是 Supervision.Resume 所做的事情。任何其他异常都被视为致命的，而且通过策略 Supervision.Stop 处理都会停下来。除了 Resume 和 Stop 以外，还有 Supervision.Restart，这是一个更细的粒度。不仅仅会从流中删除元素并恢复处理，而且处理阶段累积的状态也会被清除。Akka Streams 的文档中包含了所有细节。

7.8.2 冗余集群

使用 Akka actor 构建的模型可以被部署在没有单点故障或瓶颈的对等集群中。Akka 集群通过使用 gossip 协议实现了自动故障检测。[1] 在当前的用例中，通常借助实现器将底层 actor 作为实现技术。因此，在集群中部署基于流的模型时，获得的弹性与使用基于 actor 应用模型获得的弹性没有任何区别。因为故障处理是响应式的，所以当某个节点发生故障时，可以自动转移到其他节点——这也是弹性模型的一个重要方面。

Akka 集群是一项复杂的服务，如果想知道更多细节，请参阅 Akka 关于集群的文档。[2]

7.8.3 数据的持久化

考虑这样一个领域模型，它在内存中管理大部分数据而不是将所有内容存储在持久化存储中。对一些用例来说有其合理性——例如，为了更高的吞吐量和更少的移动部件管理。Martin Fowler 在他的网站上解释了一些此类用例以及模型设计上的变化。[3] 但一旦将数据存储在内存中，还是需要考虑其可靠性和持久性。在应用程序重启、JVM 故障或群集迁移的情况下，不能丢失任何数据。我们需要使用能够防止此类后果的技术，这也是我们讨论领域模型弹性最后的本质。在当前的用例中，Summarizer actor 维护了交易的可变状态，用内存中的 Map 架构进行结算。那么，

1 gossip是一种协议，通过该协议，集群成员彼此之间交换成员信息，确保所有成员都清楚拓扑信息。可以在Akka关于集群的文档中找到更多详细信息（http://doc.akka.io/docs/akka/snapshot/common/cluster.html）。

2 Akka集群文档请参阅http://doc.akka.io/docs/akka/2.3.11/scala/cluster-usage.html.。

3 MemoryImage，http://martinfowler.com/bliki/MemoryImage.html。

如何确保即使应用程序失败时状态也能得到维持呢？

Akka 提供了持久性 actor，它确保 actor 的所有内部状态以增量方式存储在永久存储中。一旦通过事件触发状态发生更改，事件本身将被保存在仅能追加的存储中。要注意的是整个数据不会被复制，只有事件作为日志的一部分被保留。此过程将按时间顺序保留模型在该日期之前收到的所有事件的日志。所以现在我们拥有了导致内存中数据结构当前状态的整个事件历史，可以随时重放，研究从状态 *s* 演变到状态 *t* 的过程，它是整个系统的完整审计线索。这是一种具有普遍理论背景的技术。它被称为事件溯源，并且也是使响应式模型持久化的普遍方式之一。我们将在第 8 章中将讨论事件溯源。

在当前示例中，需要做些什么改动来持久化 actor？这里有一个 `Summarizer` 建模为 `PersistentActor` 的版本。

清单 7.3 Summarizer 作为 PersistentActor

```
import akka.persistence.PersistentActor
import akka.stream.actor.ActorSubscriberMessage.OnNext
import akka.stream.actor.{ActorSubscriber, MaxInFlightRequestStrategy}

import scala.collection.mutable.{ Map => MMap }

import scalaz._
import Scalaz._

class Summarizer extends PersistentActor
  with ActorSubscriber with Logging {
  private val balance = MMap.empty[String, Balance]

  override def persistenceId = "transaction-netter"

  def receiveCommand = {
    case OnNext(data: Transaction) => persistAsync(data) { _ =>
      updateBalance(data)
    }
    case LogSummaryBalance => logger.info("Balance so far: " + balance)
  }

  def receiveRecover = {
    case d: Transaction => updateBalance(d)
  }

  def updateBalance(d: Transaction) = balance.get(d.accountNo).fold {
    balance += ((d.accountNo, Balance(d.amount, d.debitCredit)))
  } { b =>
    balance += ((d.accountNo, b |+| Balance(d.amount, d.debitCredit)))
  }
}
```

Summarizer 扩展了来自 Akka Persistence 中的 PersistentActor，它帮助我们存储富状态 actor。

需要指定唯一的 ID（跨系统唯一）。这个状态通过这个 ID 进行存储及索引，因此可以很容易从底层存储中提取出来进行重播。

需要定义 PersistentActor 发布的 receive-Command 而不是 receive。

当时间在恢复过程中被重播时，运行 receiveRecover。

除了将 `Summarizer` 定义为一个 `PersistentActor` 之外，还需要在配置文

件中指定底层存储的详细信息，Akka 将在这里存储 actor 的状态。有几个可用的日志插件，它们的具体详细信息在 http://doc.akka.io/docs/akka/2.4.4/scala/persistence.html 中有描述。

持久化一个 actor 可以防止数据丢失，将该数据存储为 actor 状态。这可以使模型在面对应用重启、系统故障和集群迁移时保持弹性。这也使系统更加具有可扩展性，因为我们已经从架构中去掉了一个重要的瓶颈：变化。我们只需要附加上更新模型内部状态的事件，而不用更新持久化存储。附加不需要加锁，所以它们比更新更容易扩展。此外，模型还会变得更容易追溯，因为现在可以在任何时间线上重播事件，并从过去的任何时间点上恢复快照。这样架构就内建了审计路径。在第 8 章中，当我们谈论事件溯源和相关技术时，将会看到这种架构模式的更多作用。

7.9 基于流的领域模型与响应式原则

我们已经讨论了使用响应式流作为主干建模领域的所有方面。现在回顾一下，看看这个架构范式是否满足我们在第 1 章开始所讨论的响应式建模以及在“响应式宣言”（www.reactivemanifesto.org）中所规定的所有原则。为了方便起见，我们复制了第 1 章中展示的响应式领域模型的 3 + 1 视图。在图 7.4 中显示了响应式流如何实现这 3 个特征。

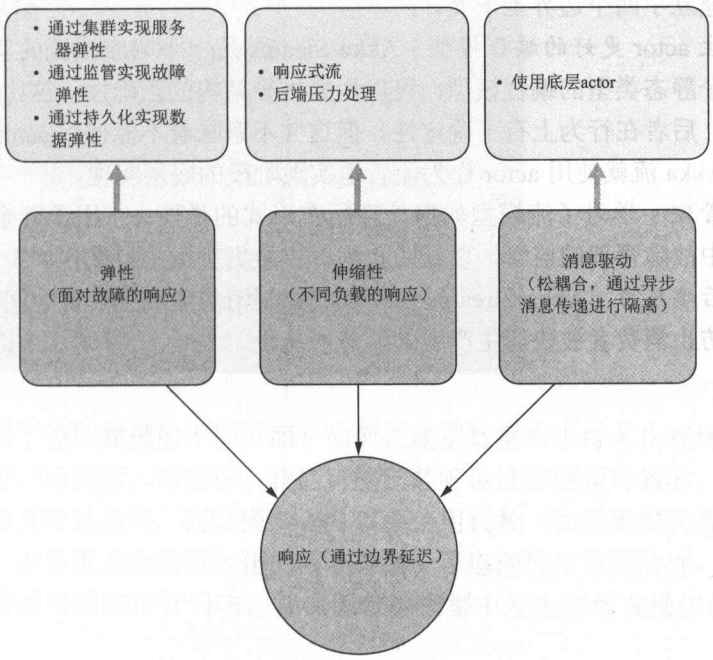


图 7.4 响应式流实现了响应式领域模型的所有特性。

图 7.4 简要地解释了响应式领域模型的各种特性。下面是如何使用基于流的实现获取端到端的响应式模型，一个具有良好响应能力的模型应具备以下 3 个特征：

- *弹性*：基于流的实现，我们看到了领域模型中 3 种形式的弹性。使用基于 actor 的监管技术，通过集中管理故障来实现故障恢复，这在响应式流中同样有效。我们通过集群获得应用程序故障的弹性。最后，Akka *Persistence* 提供数据丢失方面的弹性。
- *伸缩性*：此特性使我们能够在负载发生变化时扩展模型。正如具有后端压力处理的响应式流实现那样，可以根据每个应用的需要控制请求流和数据流。
- *消息驱动*：这是显而易见的，因为我们使用 actor 作为底层实现。

7.10 总结

本章对如何使用响应式流进行领域建模做了全面介绍。我们从一个业务用例开始，学习了生产者以及消费者的完整实现。本章的主要结论如下：

- *使用响应式流进行领域建模*：我们讨论了使用基于流的模型的原因，以及如何使用基于流的骨干来实现响应式架构。
- *处理异步边界*：响应式流是处理异步边界的一个有效实现，其上下文在空间和时间上是解耦的。我们的实现恰恰对应了这样的一个用例，在前台和后台之间建立了两个边界上下文。
- *比原生 actor 更好的编程模型*：Akka Steams 是一个响应式流的实现，它提供了一个静态类型的编程模型，可以用来推导架构的正确性。这比用 actor 会好很多，后者在行为上有不确定性。但这并不意味着不能使用 actor 作为实现方式，Akka 流就使用 actor 作为流管道实现阶段的底层实现。
- *弹性管理*：学习了流模型如何处理所有形式的弹性——用于冗余的集群，用于集中故障管理的监管，以及防止内存中数据丢失的持久化。
- *处理后端压力*：Akka Streams 帮助我们控制消费者和生产者之间的后端压力，从而防止消费者被快速生产者的流量所淹没。

8 响应式持久化与事件溯源

本章包括

- 如何将领域模型持久化到数据库中
- 持久化的两个主要模型：CRUD 模型与事件溯源模型
- 这两个模型的优缺点，以及如何选择领域模型架构
- 一个事件溯源领域模型接近完整的实现
- 如何使用函数式到关系型框架实现一个基于 CRUD 的模型

本章展示了领域建模的不同方面：如何在底层数据库中持久化领域模型，而且存储必须可靠、可回放、可查询。我们可能已经听说过存储是可靠的，它可以防止数据丢失，而且可以查询，同时提供 API 以便使用代数（如关系型代数）从数据库中查询数据。本章重点介绍持久化的两个方面：可追溯性与可回放性。在这些情况下，数据存储还可用作审计日志，并保留数据模型中发生的所有操作的完整跟踪。

想象一下提供了领域与数据模型内建可追溯性和可审计性的存储策略所带来的商业价值！

图 8.1 是本章内容的示意图。该指南有助于浏览本章时选择主题。

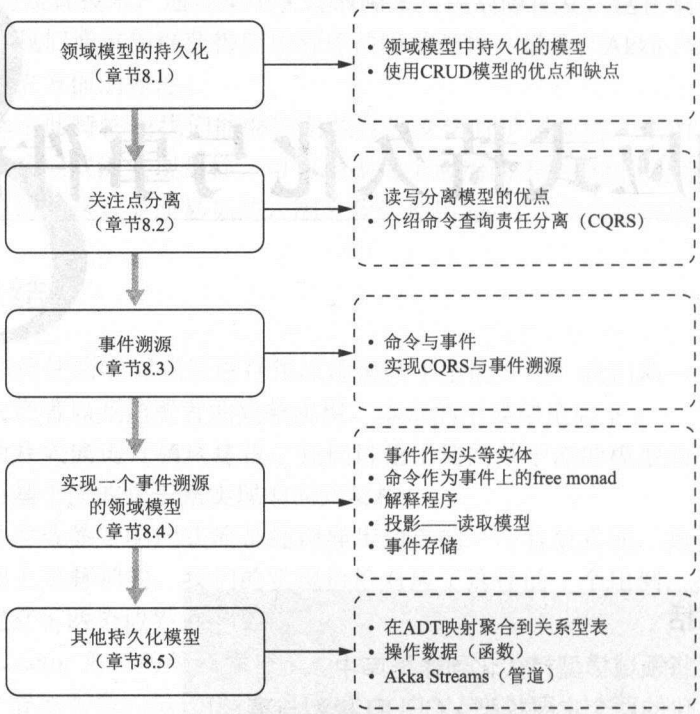


图 8.1 本章学习过程。

到本章结束时，应该能够理解如何使用适当的抽象来使领域模型对用户保持响应以及对故障保持弹性。

8.1 领域模型的持久化

第 3 章介绍了领域对象生命周期的三个主要阶段。任何领域对象都通过它的组件进行创建，参与它应该合作的各种角色，并最终保存到持久化存储中。在迄今为止的所有讨论中，我们以仓储的形式抽象了持久化，通常使用 API 来创建、查询、更新以及删除领域元素。以下是我们讨论过的 API 形式：

```
trait AccountService[Account, Amount, Balance] {
  def open(no: String, name: String, rate: Option[BigDecimal],
           openingDate: Option[Date], accountType: AccountType)
```

```
        : AccountRepository => NonEmptyList[String] \/ Account
  //..
}
```

此 API 非常明确，将一个存储作为外部组件进行注入，以便 open 能正确操作。现在是时候考虑应该在模型中使用什么形式的 AccountRepository，并在生产环境中进行部署。

在大多数模式中，存储所用的通用形式是关系型数据库管理系统。所有领域行为最终映射到一个或多个 Create、Retrieve、Update 和 Delete（CRUD）操作中。在大多数应用程序中，这是一个可以接受的模型，而且大量成功的部署就是在复杂领域模型底层使用这种数据建模架构。

使用基于 RDBMS 的 CRUD 模型作为存储的最大优势是它被广大开发人员所熟知。SQL 是最常用的语言之一，我们还有一个映射框架，它管理着 OO 或函数式领域模型与底层关系型模型之间的不匹配。[1]

但是持久化模型的 CRUD 方式至少有几个缺点。RDBMS 通常具有单点故障，并且超过一定量的数据之后就非常难以扩展，特别是在高写入负载的情况下。使用可序列化 ACID[2] 事务语义的并发写入不能扩展到单个节点以外。这不仅需要不同的数据模型思考方式，而且需要一个完全不同的范式来考虑领域模型的一致性语义。[3] 本书将讨论如何使用替代的持久化模型解决可扩展性问题。

想要理解困扰持久化 CRUD 模型的其他问题，特别是底层函数式领域模型，可以参考第 3 章讨论过的 CheckingAccount 模型：

```
case class CheckingAccount (no: String, name: String,
  dateOfOpen: Option[Date], dateOfClose: Option[Date] = None,
  balance: Balance = Balance()) extends Account
```

CheckingAccount 是一种不可变的代数数据类型，我们不能对类的实例进行任何修改。我们已经看到了不可变性的好处和共享可变性的坏处，这是本书在过去的 7 章中所讨论的纯函数建模的精华之一。现在让我们将此操作转换为基于 CRUD 的持久化模型。图 8.2 显示了对帐户执行 debit 操作时的结果。

1 或要求管理。

2 ACID就是原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation），以及持久性（Durability）。

3 放弃ACID，考虑数据（http://highscalability.com/drop-acid-and-think-about-data）。

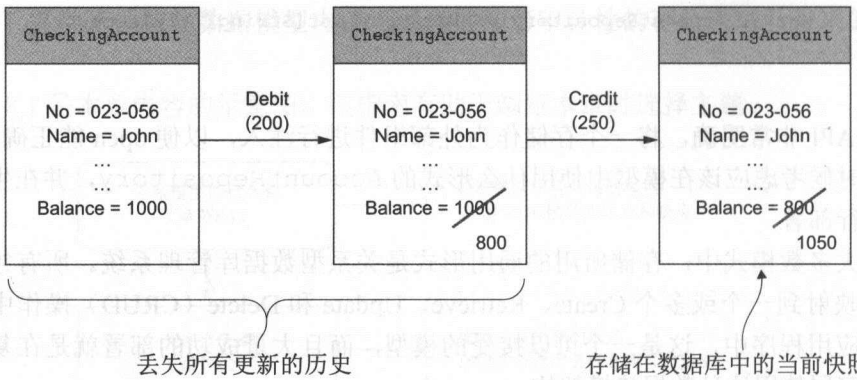


图 8.2 使用 RDBMS 在基于 CRUD 的持久化模型中更新。它存储当前快照，丢失所有更改的历史记录。

现在有了一个表 CheckingAccount，它存储了与该帐户相关的所有数据。每当通过影响帐户余额的领域行为处理一个指令时，字段 balance 将立即在空间中被更新。因此，经过一系列更新后，只会显示帐户的最新余额。我们丢失了很重要的数据：丢掉了导致当前余额为 $ 1,050 的所有操作过程。

这意味着我们将无法从底层持久化数据中执行以下任何操作：

- 查询更改的历史记录（除非去挖掘数据库存储的审计日志，这本身就是件极其复杂的事）。
- 将系统回滚到以前的某个时间进行调试。
- 从零开始重新启动系统，然后回放目前为止发生的所有事务，并使系统回到现在的状态。

总而言之，我们已经失去了系统的可追溯性，现在只有当前快照的样子，*数据模型是共享可变状态*。本章所介绍的技术允许以时间作为单独维度来建模存储。该系统从一开始就保留了完整的变更记录，并以时间先后顺序提供模型完整的可追溯性和可审计性。因为我们已经是函数式编程的专家了，可以说我们模型的当前状态是所有先前状态的一个折叠（fold），它以开始状态作为初始。

8.2 关注点分离

本章讨论持久化模型时，经常对函数式领域模型某些方面进行类比，以提醒我们回顾从中所学到的经验教训。让我们看看是否可以在持久化层面上也应用一下这些技术并享受类似的好处。关注点分离是一种交叉的质量，我们认为它是设计任何系统的理想属性。在领域模型设计的讨论中，已经看到了函数式编程是如何将模型的“*what*”和“*how*”分离开的。也看到了如何专注于设计抽象，如第 5 章中的 free

monad，我们将抽象的定义与解释程序分离开。当我们谈论领域模型的持久化时，需要清晰分离的两个方面分别是读和写。在本节中，将学习如何实现这种分离，同时使数据模型更好地模块化。

8.2.1 持久化的读 / 写模型

一个想要查看模型的用户通常希望从业务的视角去看待它。例如，查看帐户时通常希望按照业务领域所要求的格式和准则查看所有属性。这个底层的 Account 聚合可能由多个实体组成，并被标准化以便于存储。但作为一个用户来说，想看到的是基于上下文的非标准化视图。如果是银行帐户的持有人，并且想要查看帐户余额，应该能够从网上银行得到需要的所有细节。系统的另一用户可能只是对会计视角的快照感兴趣，那么她也许只对帐户的属性感兴趣，出于结算的目的她需要将这些属性发送给下游的 ERP 系统。在这里，我们讨论的是相同底层组合的两个视角，一个是网上银行上下文，另一个是结算上下文。而且这是完全独立于底层的存储模型，图 8.3 给出了这两个模型分离的概述。

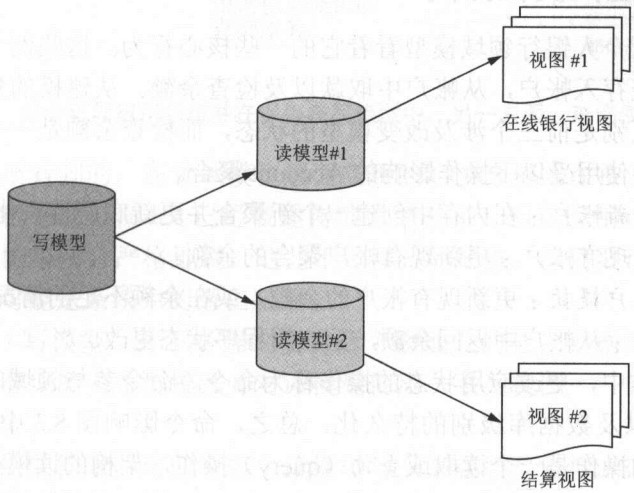


图 8.3 读 / 写模型分离。单个写模型对应多个读模型。读取内容取决于边界上下文和特定读模型所需要的信息。

所以我们已经在底层数据模型中实现了一个分离的层次。所有读取将由读取模型提供，并且所有写入都将通过写模型完成。从系统工程的角度来看，如果稍微想一下，就会发现这种分离是非常有意义的。对于读取，我们希望使底层数据更接近于它们所提供的查询和报告。这是一种避免加入关系型查询的非标准化形式，而且这与在执行写操作时所做的完全相反。因此，通过分离这两个方面，可以在读写水平获得更合适的模型。

相对来说，读取更容易扩展。在关系型数据库中，可以根据负载添加读取从库，并获得接近于线性的扩展性。而另一方面，写入则完全不同。如果使用基于 CRUD 的写模型，将会遇到共享可变状态的所有问题。与此领域模型的区别在于，RDBMS 在这里为我们管理着所有状态。但是由其他人管理并不意味着问题就不存在了，它们只是向下移了一级。在这个讨论的过程中，本书将尝试解决这个问题，并找出不必处理持久化数据变更的替代模型。

图 8.3 中的架构有很多还没有解决的问题：

- 读模型如何被写模型填充？
- 谁负责生成读模型并修复生成它们的协议？
- 前面的架构解决了什么样的扩展性问题？

我们将很快解决所有这些问题。但首先让我们来看一个在领域模型级别的模式，它将读取数据（查询）与更新聚合及应用程序状态（我们称为命令）的领域行为隔离开，并充分利用这个底层的读写持久化模型。

8.2.2 命令查询责任分离

让我们回到个人银行领域模型看看它的一些核心行为。这些例子包括：在银行开立帐户、将钱存入帐户，从帐户中取款以及检查余额。从建模的角度来看，这些行为最明显的区别是前三个涉及改变模型的状态，而检查余额是一个读操作。要实现这些行为，将使用受以下操作影响的 Account 聚合。

- *开设一个新帐户*：在内存中创建一个新聚合并更新底层持久化状态。
- *将钱存入现有帐户*：更新现有帐户聚合的余额。
- *从现有帐户提款*：更新现有帐户的余额，或在余额不足的情况下抛出例外。
- *检查余额*：从帐户中返回余额，无应用程序状态更改。

在这些操作中，更改应用状态的操作称为命令。命令参与领域的验证，更新内存中应用状态以及数据库级别的持久化。总之，命令影响图 8.3 中架构的写模型。检查帐户余额的操作是一个读取或查询（query）操作，架构的读模型可以很好地提供服务。

命令查询责任分离（CQRS）模式是一种鼓励单独领域模型各自处理命令和查询的模式。[1] 它在领域模型以及持久化模型中提供了读写分离。图 8.4 描述了 CQRS 的基本架构。

1 关于CQRS的更多信息，请参考https://msdn.microsoft.com/en-us/library/dn568103.aspx。

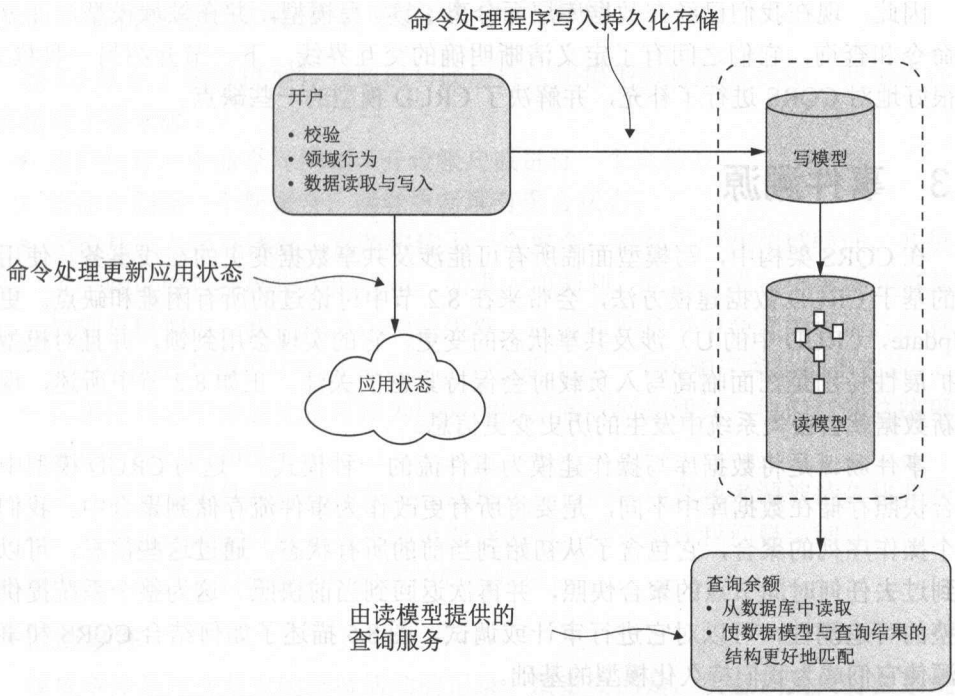


图 8.4 命令更新应用状态并在保存在写模型中，另一方面，查询使用读模型。

正如我们所看到的，在领域模型层面应用的 CQRS 模式在持久化层面很好地补充了双重模型的想法。图 8.4 已经表述得非常清楚了，但在应用 CQRS 模式时还有一些需要注意的事情。严格地说，这些不是模式的缺点，但需要仔细看看，并决定这些约束是否可能是系统中的潜在瓶颈：

- 对于读 / 写模型，并不必强制要有两个物理上的独立数据库。可以对写模型使用标准化的关系型模式并对读模型使用一系列视图——针对同一数据库的所有部分。
- 根据应用的需求，可能有多个读模型与一个底层写模型。请记住，读模型需要提供查询服务。因此，读模型的模式需要尽可能接近查询视图。这并不一定是关系型的，写模型可以是关系型的，而读模型可以由完全不同的服务（例如图形数据库）提供。
- 在某些情况下，读模型需要与写模型完全同步。这可以使用两个模型之间的发布—订阅机制或通过一些明确实现的周期性工作来完成。无论哪种方式，读 / 写模型之间都可能会出现不一致的窗口，应用需要处理最终的一致性。

因此，现在我们已经在数据库层面分离了读 / 写模型，并在领域模型层面分离了命令和查询，它们之间有了定义清晰明确的交互界线。下一节介绍另一种模式，它很好地对 CQRS 进行了补充，并解决了 CRUD 模型的一些缺点。

8.3 事件溯源

在 CQRS 架构中，写模型面临所有可能涉及共享数据变更的在线事务。使用典型的基于 CRUD 数据建模方法，会带来在 8.2 节中讨论过的所有困难和缺点。更新（Update，CRUD 中的 U）涉及共享状态的变更，它的实现会用到锁，并且对模型的可扩展性特别是在面临高写入负载时会保持紧密的关注。正如 8.2 节中所述，现场更新数据就会丢失系统中发生的历史变更信息。

事件溯源是将数据库写操作建模为事件流的一种模式。[1] 这与 CRUD 模型中将聚合快照存储在数据库中不同，是要将所有更改作为事件流存储到聚合中。我们有整个操作序列的聚合，它包含了从初始到当前的所有状态。通过这些信息，可以回滚到过去任何时间节点的聚合快照，并再次返回到当前快照。这为整个系统提供了完整的可追溯性，可以对它进行审计或调试。图 8.5 描述了如何结合 CQRS 和事件溯源使它们成为我们持久化模型的基础。

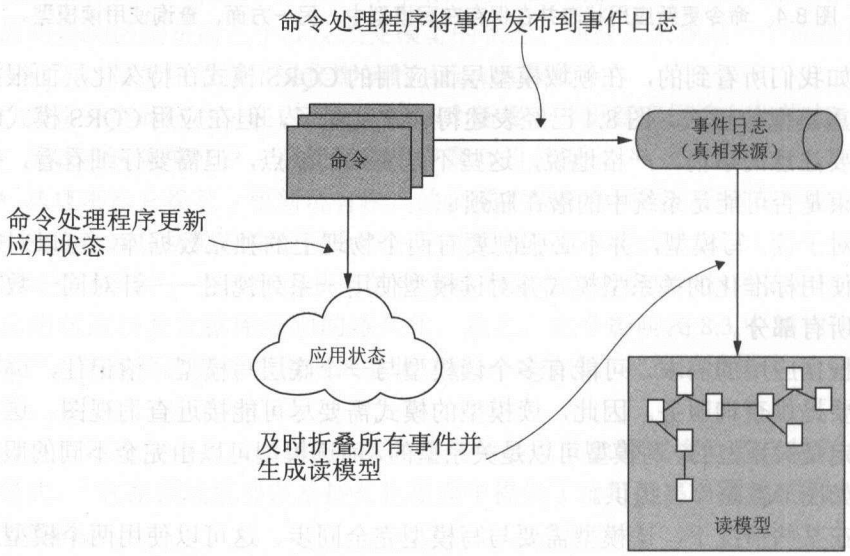


图 8.5 事件作为真相来源，它被折叠生成系统的当前快照。这会生成用于查询的读模型。

1 关于事件溯源的更多信息，请参见https://msdn.microsoft.com/en-us/library/dn589792 .aspx。

8.3.1 事件溯源领域模型中的命令和事件

图 8.5 展示了领域模型行为如何与底层持久化模型进行相互作用，使用事件作为真相的主要来源：

- 用户执行一个命令（例如，开设帐户或进行一个交易）。
- 该命令创建一个新聚合，或者更新现有聚合状态。
- 无论是以上哪种情况，它都构建了一个聚合，进行了一些领域验证，并执行了一些可能包括副作用的操作。如果一切顺利，它会生成一个事件并将其添加到写模型中。我们称之为领域事件，在后面的章节中会详细讨论。注意我们并不更新写模型中的任何东西，它本质上是一个按顺序增长的事件流。
- 在事件日志中添加记录可以为感兴趣的订阅者创建通知。他们订阅这些事件并更新他们的读模型。

总之，事件是事件溯源模型中的关注焦点。它们作为真相来源被持久化并向下游发布，对其感兴趣的订阅者将用来更新读模型。接下来让我们更详细地讨论什么是领域事件，以及它们在模型中扮演的确切角色。

领域事件

领域事件是过去发生的领域活动的记录。最重要的是，要意识到只有在系统中某些活动已经发生之后才会生成一个事件。所以不能改变一个事件，否则，就是在改变系统的历史。以下是领域模型中事件的几个定义特征：

- *通用语言*：事件必须有一个名称，它是领域词汇的组成部分（就像任何其他领域模型组件一样）。例如，要表明已为客户开设了一个帐户，可以将事件命名为 `AccountOpened`。或者，如果有合适的模块命名空间，就可以有一个名为 `AccountService` 的模块，里面有一个名为 `Opened` 的事件。还要注意的是，一个事件的名称必须是过去时态，因为它建模的是过去已经发生的事情。
- *触发*：命令的执行或其他事件的处理都可能生成一个事件。
- *发布—订阅*：兴趣方可以订阅特定的事件流，并更新其各自的读模型。
- *不可变*：不可变的事件通常被建模为代数数据类型。当我们讨论到某个用例时，就会看到具体实现。
- *时间戳*：事件是在特定时间点发生的。任何对事件建模的数据结构中必须包含一个时间戳。

图 8.6 描述了一用例：执行作为帐户服务一部分的特定命令，事件被记录到写模型中。这个简单的用例解释了命令如何生成事件的基本概念，而事件被记录到写模型中并帮助订阅者更新其读模型。在此特意选择了一个我们都很熟悉的用例，以

便可以和前几章中学过的例子相关联。

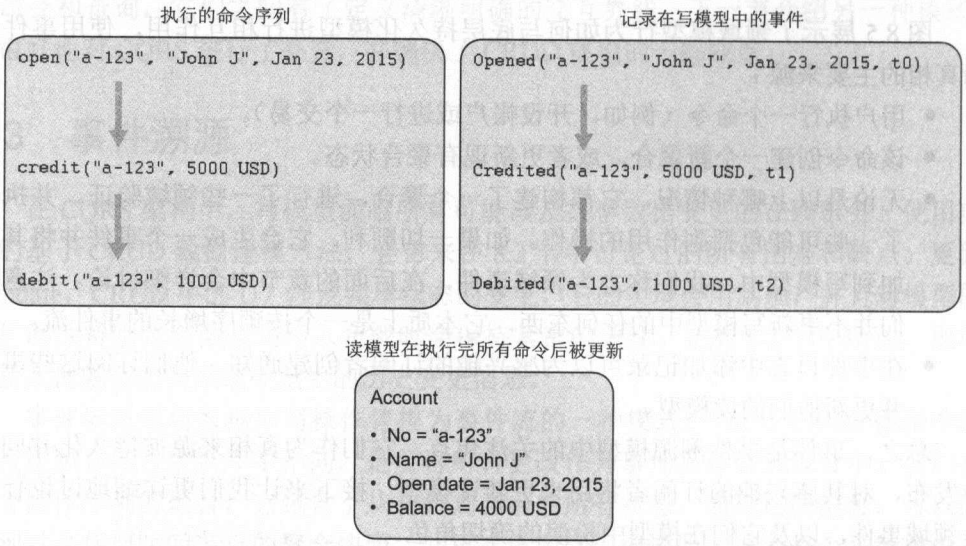


图 8.6 命令生成的事件被记录进事件日志中（写模型）。各方可以订阅事件流并更新读模型。请注意领域事件的命名以及它们如何属于领域词汇表。

命令处理程序

命令是一些动作，它们也体现了事件的双重性。在实现的层面，当抽象一个命令可以执行的领域行为时，我们讨论命令处理程序。这里包括创建或更新聚合、执行业务校验、添加到写模型以及与外部系统进行任意次数的交互，例如发送电子邮件或将消息推送到消息队列。我们知道如何抽象副作用，将使用相同的技术来实现命令处理程序。命令处理程序仅仅只是抽象，我们对其进行评估，用来执行一系列会生成事件的操作并被添加到流中（写模型）。图 8.7 描述了命令处理程序中的操作流，它实现了从客户帐户进行借款的功能。

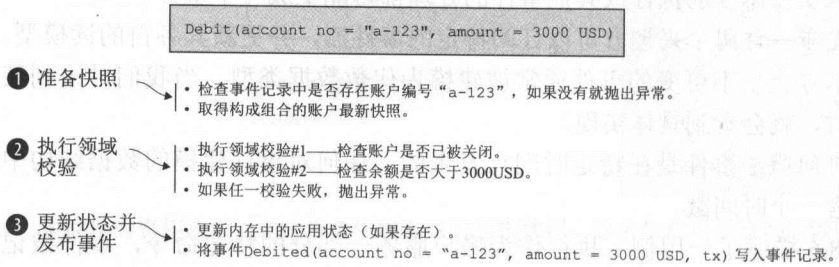


图 8.7 从客户帐户借款的命令处理程序，注意命令处理程序中流所涉及的 3 个主要阶段。

8.3.2 实现 CQRS 和事件溯源

如本章前文所述，事件溯源和 CQRS 通常一起被用作领域模型中的持久化模式。图 8.6 展示了当命令被处理并生成事件时所执行的流程案例。但是一些相当微妙的点使实现模型并不那么简单，特别是如果想保持函数式模型以及引用透明。本节将接着上节中的实现案例，介绍其中的一些问题。

作为回顾，下面是通过 CQRS 和 ES 增强的领域模型的命令流和事件流，以及附加的详细信息：

- 通过服务接口分发命令（接口可以是 UI 或 REST 端点，或任何调用领域服务的代理）。
- 如果命令需要与聚合的新实例一起工作（如创建一个新账户），就在事件存储中检查该实体是否还没有被创建。如果校验失败则报异常。
- 如果命令需要与一个已经存在的聚合一起工作（如余额更新），就需要通过回放事件存储中的事件，或从快照存储中获取最新版本的快照来创建实例。要注意的是，快照存储是一种可选优化，它通过从事件存储中定期回放来存储每个聚合的最新快照。但对于大型领域模型来说，使用快照存储主要是出于性能的考虑。
- 命令处理器对聚合和输入参数执行所有业务校验并继续处理命令。注意，命令处理涉及两个主要步骤：
  - 执行改变聚合状态的领域逻辑，例如在帐户关闭命令时设置帐户关闭日期，并且可能会涉及副作用（例如发送电子邮件或与第三方服务交互）。
  - 生成要记录到事件存储中的事件。作为响应，特定事件的订阅者将更新它们的读模型。
- 如果需要从头重建特定聚合，则必须从事件日志中回放事件。这听起来很简单，但是记住，回放事件只需要重新创建状态，而不用重复之前执行的命令所具有的任何副作用。本书将在实现模型中讨论如何从状态变更（命令和事件处理程序都需要执行）中解耦副作用（只有命令处理程序需要执行）。

批处理命令

我们一直很重视可组合性，为什么要因为命令而放弃它呢？毕竟命令也是 API，我们希望它们也是可组合的，可以通过较小的命令组成更大的命令，正如在前面章节讨论 API 的组合那样。这里是我们领域中的一个例子。假设有两个单独的账户 `debit` 和 `credit` 命令，那么这两者组合成的 `transfer` 命令可能是这样的：

```
def transfer(from: String, to: String,
  amount: Amount): Command[Unit] = for {
    _ <- debit(from, amount)
    _ <- credit(to, amount)
  } yield ()
```

记住，在第 5 章中领域服务 API 具有类似的组合性。还记得我们所使用的技巧吗？没错！就是 free monad。[1] 在事件溯源领域模型的实现中它要东山再起了。将命令定义为事件上的 free monad，这样就可以单子化地组合命令来构建模型行为的代数。我们将在下一节中了解所有详细信息。

处理副作用

如果想保持模型的纯粹性以及引用透明，有一个极其重要的问题需要解决，就是如何处理副作用。执行命令处理程序会产生副作用，我们希望将它们从改变状态的 API 中解耦出来。事件处理程序在回放时需要改变状态，而且不会诱发副作用。我们一定不希望客户在每次回放模型事件时都收到电子邮件。

为命令组合采用 free monad 之后，free monad 的解释程序可以处理所有的副作用。并且因为命令是 free monad，副作用被限制于命令的执行中，所以必须将改变状态的 API 实现为独立函数，这样它就可以被命令和事件处理程序单独地重用。这只是我们将采用的策略的概述，下一节中的实现提供了所有的细节。

8.4 实现事件溯源的领域模型（函数式）

本节还是借用个人银行领域，来讲述事件溯源领域模型的一些实现内容。我们使用了前几章中的一些用例，这样就可以关联到相关功能，并用事件溯源的范例来考虑那些模型元素的持久化。这绝不意味着它是一个可用于生产环境的事件溯源领域模型的实现，我们尽可能地简化了实现，只是为了提供一个函数式实现的思路，并突出一些可能导致的相关问题。

如上一节所提到的，我们将使用 free monad 来实现事件溯源。这不是实现事件溯源的唯一方法（还有许多其他替代技术），甚至在函数式编程中也有其他方法。但这里选择这种方法主要是因为它在解释程序中提供了很好的事件代数分离。我们可以选择继续构建纯粹的命令序列，并且只有当我们有最终的命令栈时，才可以通过代数解释程序提交赋值。这使得整个模型更具有组合性，并且将副作用限制在解释程序内部——这也正是我们对 FP 所要求的。

以下是我们大致的实施策略：

1 参见第5章中与free monad相关的章节。

- 通过将每个事件定义为可以彼此连接的代数数据类型来定义事件的代数。当我们要实现一个基于 free monad 的帐户存储时，第 5 章中有讲到如何实现这一点。
- 在事件外生成一个 free monad，这个 monad 就是命令。所以现在可以通过使用 for 表达式来组合命令并构建更大的命令。这是我们实现的第一阶段，它将我们的命令堆栈建立为一个没有外延的纯数据类型。
- 在实现的第二阶段中，执行与每个命令相对应的操作。在这里我们设计了一个解释程序，它接受整个命令栈，遍历它并对与每个命令相关的事件进行模式匹配。对于每个事件，我们实现了为处理命令而要执行的操作。

使用 free monad 的事件溯源——代数解释程序

命令是构建纯数据类型的 free monads，之后使用事件代数来解释数据类型并执行命令。

8.4.1 作为头等实体的事件

领域事件是事件溯源整个范式中的头等实体。整个模型都是基于领域事件的，模型历史被存储为事件流，并且通过事件回放来完成模型（重新）生成。我们在领域模型的实现中反映出同样的事实并不奇怪。清单 8.1 提供了基本的 `Event` 抽象以及事件的代数，这些都计划在 `AccountService` 实现中进行处理。

清单 8.1 事件的代数

```
import org.joda.time.DateTime
import cqrs.service._
import common._

trait Event[A] {
  def at: DateTime
}

case class Opened(no: String, name: String,
  openingDate: Option[DateTime],
  at: DateTime = today) extends Event[Account]

case class Closed(no: String, closeDate: Option[DateTime],
  at: DateTime = today) extends Event[Account]

case class Debited(no: String, amount: Amount, at: DateTime = today)
  extends Event[Account]
```

事件的基本抽象

事件触发的时间

```
case class Credited(no: String, amount: Amount, at: DateTime = today)
  extends Event[Account]
```

所有事件的完整代数，我们想将它们作为用例的一部分来处理，此部分完全针对 Account 聚合。

这里，所有事件都以过去时态命名，这表明它们已经发生过。因此，它们是不可变的，正如可以从为它们定义的每一个代数数据类型进行推断一样。最后，事件的名称来自于领域词汇，这也许是我们目前为止已经学会并接受的规范。

现在，我们已经知道为客户提供帐户服务的每个事件都长什么样了，让我们回过头来，首先考虑一下这些事件是如何生成的。正如 8.3.2 节所述，命令得到执行，聚合的状态在这些动作的过程中发生变更，如果命令成功执行，则生成一个事件。在该章节中聚合状态还可能在事件回放过程中被更改。让我们看看如何建模 API，才能在这两种情况下影响聚合状态的变化。

由于改变聚合状态的根本目的是生成其当前的快照，我们将在名为 Snapshot 的模块中定义两个函数。清单 8.2 介绍了一些基本的常见类型，并定义了 Snapshot 模块。

清单 8.2 聚合状态变更与生成快照的 API

```
import scalaz._
Import Scalaz._

object Common {
  type AggregateId = String
  type Error = String
}

import Common._

trait Aggregate {
  def id: AggregateId
}

trait Snapshot[A <: Aggregate] {
  def updateState(e: Event[_], initial: Map[String, A]): Map[String, A]

  def snapshot(es: List[Event[_]]): String \/ Map[String, A] =
    es.reverse.foldLeft(Map.empty[String, A]) { (a, e) =>
      updateState(e, a) }.right
}
```

一些基本类型定义

聚合的基本模块。它必须有一个并且是唯一的 ID。

根据事件更新聚合状态的函数。它从事件中获取聚合 ID，从提供给它的滚动快照中获取当前状态，并根据事件更新状态。请注意，该函数被指定到聚合，且必须为每个聚合分别单独定义。

通用快照函数。获取事件列表，从空状态开始回放它们，并提供事件列表中包含的所有聚合的快照。

在清单 8.2 中，函数 updateState 类似于 State monad。[1] 它需要一个初始状

1 在4.2.3节中已经广泛地讨论了State monad。

态和一个 Event，并将聚合的状态更新为当前快照。updateState 中的处理取决于集合，它需要作为特定聚合功能的一部分来被实现。函数 snapshot 是通用的，它只是提供事件列表的一个折叠，并为所有参与的聚合生成当前快照。

下一个步骤是了解 updateState 是如何为我们工作的，如清单 8.3 所示。[1]

清单 8.3 用于 Account 聚合的状态变更 API

```
object AccountSnapshot extends Snapshot[Account] {
  def updateState(e: Event[_], initial: Map[String, Account])= e match {
    case o @ Opened(no, name, odate, _) =>
      initial + (no -> Account(no, name, odate.get))

    case c @ Closed(no, cdate, _) =>
      initial + (no -> initial(no).copy(dateOfClosing =
        Some(cdate.getOrElse(today))))

    case d @ Debited(no, amount, _) =>
      val a = initial(no)
      initial + (no -> a.copy(balance =
        Balance(a.balance.amount - amount)))

    case r @ Credited(no, amount, _) =>
      val a = initial(no)
      initial + (no -> a.copy(balance =
        Balance(a.balance.amount + amount)))
  }
}
```

每个事件都会在 Map 中更新聚合状态。

我们已经完成大量的事件溯源领域模型实现，让我们来看看这一章节中关于实现方面的几点总结：

- 定义用于处理帐户的事件代数（清单 8.1）。
- 定义了一个状态变更 API，它可以用来根据事件（清单 8.2）改变聚合状态。当实现命令处理程序时将看到如何使用它。
- 定义了一个用于快照的 API，它使用了状态变更 API（清单 8.3）。这让我们了解如何使用快照重新创建一个应用领域模型的特定状态。例如，可以选取两个日期之间的事件列表，并要求模型准备一个快照。注意，因为我们拥有系统曾处理过的整个事件流，所以才能这样做。

在下一节中，将进入更有趣的部分：命令处理。

8.4.2 命令是事件上的 free monad

每个命令都有一个处理程序来执行命令应该执行的操作。在一次成功执行中，

1 将Account用作一个聚合。简单起见，这里没有包含Account的定义。具体详情可以在第8章的代码库中找到。Account的具体定义对于理解状态如何改变来说并不是非常重要。

事件会被发布在事件日志上。因此命令执行操作，我们将一个命令建模为一个事件上的 free monad。命令处理循环将按以下方式建模：

- 命令是一个纯数据类型。因为它是一个 monad，可以通过组合命令来形成更大的命令，构建的组合命令仍然是没有语义或操作的抽象。
- 向数据类型添加语义时，命令处理的所有作用都发生在 free monad 的解释程序中。使用事件的代数来查找适当的命令处理程序，并执行操作发布事件。

如果这听起来很复杂，不要急，马上就会看到实现。如果已经理解第 5 章中关于 free monad 的解释，那也可以在这里发现相同的概念。在这期间，会发现纯函数式事件溯源领域模型实现的优点。清单 8.4 提供了 Command 的基本模块。

清单 8.4 Command 是一个 free monad

```
trait Commands[A] {
  type Command[A] = Free[Event, A]
}

trait AccountCommands extends Commands[Account] {
  import Event._
  import scala.language.implicitConversions

  private implicit def liftEvent[A](event: Event[A]): Command[A] =
    Free.liftF(event)

  def open(no: String, name: String,
    openingDate: Option[DateTime]): Command[Account] =
      Opened(no, name, openingDate, today)

  def close(no: String, closeDate: Option[DateTime]): Command[Account] =
    Closed(no, closeDate, today)

  def debit(no: String, amount: Amount): Command[Account] =
    Debited(no, amount, today)

  def credit(no: String, amount: Amount): Command[Account] =
    Credited(no, amount, today)
}
```

Command 是一个与 Event 相关的 free monad。

这个隐式函数将一个 Event 提到 free monad 的上下文中。至于它如何帮助构建组合命令，更多详细信息请参阅文中内容。

❶ Command 的例子，注意，我们发出一个 Event，它是一个没有任何语义或命令处理逻辑的数据类型。隐式函数 liftEvent 将这个事件 Opened 提到 free monad 的上下文中。因此，我们可以拥有返回类型 Command[Account]，当脱糖时，它就是 Free [Event, Account]，而这就是 free monad。

清单 8.4 中很直观地定义了几个命令，诸如 open、close，并且每个都映射到一个 Event 类型，当接受它的解释程序时它将在执行成功后进行发布。还有一个隐式函数 liftEvent，它将 Event 提到 free monad 的上下文中。这就解释了为什么每个命令都会成为类型 Command[Account] 的 free monad。图 8.8 用一个简单

示意图解释了这种转换。

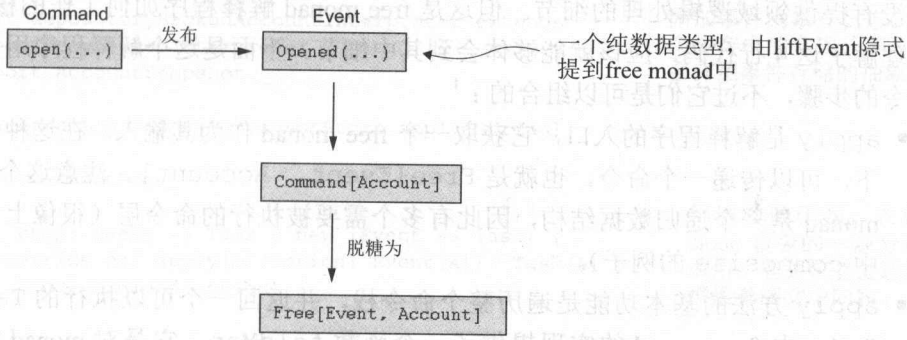


图 8.8 将一个命令建模为纯数据类型，它是所发布事件的 free monad。

借助于 monad 的优势，可以用代数方法组合命令形成更大的命令：

```
val composite =
  for {
    a <- open("a-123", "John J", Some(today))
    _ <- credit(a.no, 10000)
    _ <- credit(a.no, 30000)
    d <- debit(a.no, 23000)
  } yield d
```

这里的 composite 仍然是一个纯数据类型。我们将在下一节中看到如何从组合命令中剥离 monad 层，提取隐藏在下面的事件，解释事件的代数，并为每个事件分配动作。

8.4.3 解释程序——隐藏所有有趣的东西

作为 free monad 的专家，我相信大家已经知道，到目前为止处理有副作用的命令的理想场所是 free monad 的解释程序，通过组合构建命令并最终提交给解释程序执行实际动作。这里有一个简单的解释程序，它将调度每个命令发布的事件，并执行所有必要的领域逻辑。清单 8.5 就是一个这样的解释程序。

清单 8.5 free monad 解释程序

```
import scalaz._
Import Scalaz._
import scalaz.concurrent.Task
trait RepositoryBackedInterpreter {
  def step: Event ~> Task

  def apply[A](action: Free[Event, A]): Task[A]
    = action.foldMap(step)
}
```

解释程序获取整个 free monad（组合命令），逐步遍历组合的递归结构，并在每一步执行该步骤函数。step 函数是命令处理程序将被调用的地方，它需要针对每个聚合类型单独实现。

让我们来谈谈这个解释程序是如何工作的。实现看起来非常简洁，这是因为我们还没有提供领域逻辑处理的细节。但这是 free monad 解释程序如何工作的核心，如果理解了这 4 行代码，应该就能够体会到其中细节。下面是这个解释程序用来处理命令的步骤，不过它们是可以组合的：[1]

- `apply` 是解释程序的入口，它获取一个 free monad 作为其输入。在这种情况下，可以传递一个命令，也就是 `Free[Event, Account]`。注意这个 free monad 是一个递归数据结构，因此有多个需要被执行的命令层（很像上一节中 `composite` 的例子）。
- `apply` 方法的基本功能是遍历整个命令栈，并返回一个可以执行的 `Task`。Scalaz 中 free monad 的实现提供了一个函数 `foldMap`，它是对 monad 的折叠并为每一层映射了 `step` 函数。`step` 函数返回一个 `Task`，它依然是一个 monad。[2] 因此，`foldMap` 生成的所有 `Task` 被 *flatten* 到一起组成最终的、`apply` 所返回的 `Task`。[3]
- `step` 函数返回一个自然转换，正如第 5 章中所述，它是一个保留结构的转换。在这里根据 `Event` 构建一个 `Task`。最后，解释程序返回一个 `scalaz.concurrent.Task`，可以使用其支持的某个策略进行赋值。

交付账户服务的解释程序

现在进入到实现的最后部分，在这里需要执行所有领域行为，触发命令处理程序，运行副作用，并生成事件。听起来事情很多，但实现很简单。到目前为止，我们已经将代码组织为纯模块，甚至命令（至少到目前为止）都是纯数据类型。我们很快就会将这些命令赋予语义并作为特定帐户解释程序的一部分。清单 8.6 仅给出了构建解释程序的大体框架，可以在代码库中找到完整可运行版本。

---

1 要了解解释程序的工作原理，就需要掌握Scalaz中free monad的实现方式。这部分内容被包含在5.5.5节中。

2 `Task`是一个monad，请参阅https://github.com/scalaz/scalaz中的Task源码。

3 记住，monad提供了一个`flatMap`，它结合了`map`和`flatten`的功能。这里所有生成的`Task`通过`flatMap`形成最终的`Task`。

清单 8.6 处理命令的领域逻辑

```
object RepositoryBackedAccountInterpreter extends
  RepositoryBackedInterpreter {
  import AccountSnapshot._

  val eventLog = InMemoryEventStore.apply[String]

  import eventLog._

  val step: Event ~> Task = new (Event ~> Task) {
    override def apply[A](action: Event[A]): Task[A] =
      handleCommand(action)
  }

  private def validateClose(no: String, cd: Option[DateTime]) = for {
    l <- events(no)
    s <- snapshot(l)
    a <- closed(s(no))
    _ <- beforeOpeningDate(a, cd)
  } yield s

  // all other domain validation functions ..

private def handleCommand[A](e: Event[A]): Task[A] = e match {

  case o @ Opened(no, name, odate, _) => Task {
    validateOpen(no).fold(
      err => throw new RuntimeException(err),
      _   => {
        val a = Account(no, name, odate.get)
        eventLog.put(no, o)
        a
      }
    )
  }

  case c @ Closed(no, cdate, _) => Task {
    validateClose(no, cdate).fold(
      err => throw new RuntimeException(err),
      currentState => {
        eventLog.put(no, c)
        updateState(c, currentState)(no)
      }
    )
  }

// all other command handlers ..

}
```

附加事件的事件日志。这是一个内存中的实现。下一节中将会介绍事件存储的抽象。

领域校验函数，它作为 close 命令的一部分而被调用。有关校验处理的详情请参阅文中内容。 ❶

❷ 在事件日志中寻找聚合。

调用校验函数

❸ 准备它的快照。

命令处理的核心函数。注意它是如何调度事件以及在聚合上调用特定的领域行为的。 ❹

RepositoryBackedAccountInterpreter 的 3 个主要部分如下：

- 事件日志：在此处定义事件日志。在我们的用例中，它只是一个简单的内存中的 Map（可以参考代码库中的源码）。但是我们有一个通用接口，而且对相同的接口可以有多个实现。事实上，代码库中有一个实现，它存储了 JSON 而不是事件对象。
- 领域校验：命令处理程序的一个重要方面是执行领域校验，并将校验定义为解释程序的一部分。在这里展示了一个校验函数，不过它只提供一个思路，就是关于如何用单子作用从事件日志 ❶ 中查找聚合，准备快照 ❷，然后对其调用校验函数 ❸。
- 命令处理程序：所有命令作为事件代数解释程序的一部分被执行。清单 8.5 中 RepositoryBackedInterpreter 的 apply 方法是命令处理程序的入口。它获取整个命令栈，遍历结构，并将每个事件传递给 handleCommand 函数。注意，它传递的事件是命令在构建 free monad（清单 8.4 中的 ❶）时所使用的。所以我们从命令开始，构建一个事件上的 free monad，然后对它们进行组合，形成组合命令，接着命令处理程序通过解释事件（清单 8.6 中的 ❹）来执行这些命令。它使用事件代数并进行模式匹配，然后完成校验，同时处理程序将事件写入事件日志。最后，命令处理程序使用 updateState 这个 API，通过应用当前事件来更新聚合（Account）的状态。

运行一个例子

让我们把所有内容绑在一起，看一个运行的例子。清单 8.7 已被手动格式化以适合这些页面，不过当运行本书代码库中的代码时，也将得到类似的结果。本节已经涵盖了一部分实现，以突出其函数式的工作方式。可以从代码库中找到完整的可运行实现。

在代码库中，会看到更多使实现模块化及可重用的抽象，也会看到实现被拆分成多个可以在任何聚合中被重用的通用组件。同时还将有一些帐户的特定逻辑（例如，仅适用于帐户的特定命令和事件）。

清单 8.7 运行一个例子

```
scala> import frdomain.ch8.cqrs.service._
scala> import Scripts._
scala> import RepositoryBackedAccountInterpreter._

scala> RepositoryBackedAccountInterpreter(composite)
res0: scalaz.concurrent.Task[Account] = scalaz.concurrent.Task@2aaa2ff0
```

一个组合命令的例子，它是一个由解释程序进行解释并生成 Task 的 free monad

```
scala> res0.unsafePerformAsync
res1: Account = Account(a-123,debasish ghosh,2015-11-
    22T12:00:09.000+05:30,None,Balance(17000))

scala> for {
     |   a <- credit("a-123", 1000)
     |   b <- open("a-124", "john j", Some(org.joda.time.DateTime.now()))
     |   c <- credit(b.no, 1200)
     | } yield c
res2: scalaz.Free[Event,Account] = Gosub(..)

scala> RepositoryBackedAccountInterpreter(res2)
res3: scalaz.concurrent.Task[Account] = scalaz.concurrent.Task@3bb6d65c

scala> res3.unsafePerformSync
res4: Account = Account(a-124,john j,2015-11-
    22T12:01:23.000+05:30,None,Balance(1200))

scala> eventLog.events("a-123")
res5: scalaz.\/[Error,List[Event[_]]] = \/-(List(Credited(a-123,1000,
➥ 2015-11-22T12:00:09.000+05:30,<function1>), Debited(a-123,23000,
➥ 2015-11-22T12:00:09.000+05:30,<function1>), …

scala> eventLog.events("a-124")
res6: scalaz.\/[Error,List[Event[_]]] = \/-(List(Credited(a-124,1200,
➥ 2015-11-22T12:00:09.000+05:30,<function1>), Opened(a-124,
➥ john j,Some(2015-11-22T12:01:23.000+05:30),2015-11-
    22T12:00:09.000+05:30,<function1>)))

scala> import AccountSnapshot._
import AccountSnapshot._

scala> import scalaz._
import scalaz._

scala> import Scalaz._
import Scalaz._

scala> res5 |+| res6
res7: scalaz.\/[Error,List[Event[_]]] = \/-(List(Credited(
➥ a-123,1000,2015-11-22T12:00:09.000+05:30,<function1>),
➥ Debited(a-123,23000,2015-11-22T12:00:09.000+05:30,<function1>), …

scala> res7 map snapshot
res8: scalaz.\/[Error,scalaz.\/[String,Map[String,Account]]] =
➥ \/-(\/-(Map(a-124 -> Account(a-124,john j,2015-11-
    22T12:01:23.000+05:30,None,Balance(1200)), a-123 -> Account(a-123,
➥ debasish ghosh,2015-11-22T12:00:09.000+05:30,None,
➥ Balance(18000)))))

scala> eventLog.allEvents
res9: scalaz.\/[Error,List[Event[_]]] = \/-(List(Credited(a-123,
➥ 1000,2015-11-22T12:00:09.000+05:30,<function1>),
➥ Debited(a-123,23000,2015-11-22T12:00:09.000+05:30,<function1>), …
```

运行任务，通过一些借贷组合命令创建账户快照

创建另一个复合命令，它将创建另一个帐户，对其以及在上一步中创建的帐户执行一些操作

从事件日志中获取这两个帐户的事件集

注意事件的返回类型是 scalaz.\/，这是一个 monoid。可以将它们进行组合并得到汇总的事件集

现在，可以对这组事件运行快照，并获取这两个帐户的最新快照

```
scala> res9 map snapshot
res10: scalaz.\/[Error,scalaz.\/[String,Map[String,Account]]] =
➥ \/-(\/-(Map(a-124 -> Account(a-124,john j,
➥ 2015-11-22T12:01:23.000+05:30,None,Balance(1200)),
➥ a-123 -> Account(a-123,debasish ghosh,
➥ 2015-11-22T12:00:09.000+05:30,None,Balance(18000)))))
```

还可以一次性从事件日志中获取所有事件，并在完整集上运行一次快照。想借助事件日志中的事件流从头开始构建聚合快照时，它将能发挥重要的作用

8.4.4 投影——读取端模型

到目前为止，我们已经了解了如何处理命令并发布事件到事件日志。但是，如何实现模型需要提供的查询和报告服务？8.3 节中介绍了专门为此目的设计的读模型。读模型也被称为投影，其本质上是从写模型（事件日志）到某种查询形式间的映射。例如，如果是从关系数据库中提供查询服务，则表结构需要适当地规范化，以避免连接以及其他昂贵的操作，同时在数据模型中提供通俗易懂的查询。

建立投影无非是生成一个合适的快照函数，它能够读取事件流并更新当前模型，每个在事件日志中生成的新事件都要完成此更新。在决定如何使用读模型之前，还需要考虑投影架构的两个方面。

- *推送还是拉取*：可以让写入端推送新事件来更新读模型。或者，读模型可以按特定时间间隔拉取数据，检查是否有任何新事件需要消费。这两种模型都有其各自的优点和缺点。如果事件生成的速度比较慢而且有间歇，拉取模型可能就会产生比较多的浪费，读取端可能会在空白流上浪费大量的拉取循环。而推送模型在此时是高效的，事件只有在它们被生成时才能被推送。但拉取模型具有内置的后端压力处理，读取端拉取取决于它的消费速度。而推送模型则需要一个明确的后端压力处理机制，正如在第 7 章中提到的 Akka Streams。
- *重新启动投影*：在某些情况下，可能需要更改读模型的模式，比如写模型中的事件结构发生了更改。在这种情况下，需要更新模型的版本，同时还要最小化对查询 / 报告服务的影响。这是一个棘手的情况，具体的策略可能取决于读模型的数据量。一个策略是从事件日志开始启动一个新的投影，同时旧的投影仍然提供查询服务。在新的投影模型更新了写模型中的所有事件之后，再切换到新的投影来提供服务。

8.4.5 事件存储

事件存储是迄今为止用于系统中所有被创建的领域事件的核心存储介质。我们需要进行充分考虑，选择一个合适的存储，来满足模型的可扩展性以及性能标准。幸运的是，这种存储的语义比关系型数据库简单得多。这是因为要执行的唯一写操作就是附加。有很多只能附加的可扩展存储可供我们选择，包括 Event Store（https:// geteventstore.com），或者一些 NoSQL 存储，例如 Redis（http://redis.io）和 Cassandra（http://cassandra.apache.org）。

事件存储中的存储性质很简单。我们需要存储由聚合 ID 索引的事件。在此类存储中应该至少具备以下操作。

- 获取特定聚合的事件列表。
- 将特定聚合的事件放入存储中。
- 从事件存储获取所有事件。

清单 8.8 显示了此类事件存储的通用框架。

清单 8.8 事件存储的简单框架

```
import scalaz.\/
trait EventStore[K] {
  def get(key: K): List[Event[_]]
  def put(key: K, event: Event[_]): Error \/ Event[_]
  def events(key: K): Error \/ List[Event[_]]
  def allEvents: Error \/ List[Event[_]]
}
```

本书的代码库包含了一个事件存储的几种实现。事件存储是一个可以建立或打破模型可靠性的核心组件。在我们的例子中采用的是基于线程安全的并发 Map 的简单实现，但在现实中，需要选择能保证高可靠和高可用的实现。在大多数用于生产环境的实现中，在声明数据已经被安全地持久化之前，要使用配额将写操作 fsync[1] 到特定数量的存储中。某些实现还在事件存储的事务中提供 ACID 语义。我们可以拥有长期的事务，并将整个事务当做原子事务来对待，就像在 RDBMS 中一样。

8.4.6 分布式 CQRS——一个短信

事件溯源和 CQRS 依靠事件记录和回放来管理应用程序状态。事件存储的复制通常需要跨越多个位置或单个位置中的多个节点，甚至在单个节点上还要进行处理。此外，我们可能有多个事件存储，每个事件存储具有不同的事件日志，通过复制组合在一起以形成组合应用状态。所有复制必须是异步的，因此需要一个策略来解决

1 将文件同步到存储设备的函数。——译者注

更新时的冲突。如果有一个领域模型需要实现分布式领域服务，而这个领域服务依靠多个事件存储的协作来实现一致性，那么请务必选择具备此类功能的 CQRS 框架。

这些框架使用保存了事件之间的事前关系（因果关系）协议来复制各个位置的事件。通常来说，它们使用基于向量时钟的实现来确保因果关系的跟踪。其中一个框架就是 Eventuate（http://rbmhtechnology.github.io/eventuate），它是一种基于 Akka Persistence 的事件溯源实现（http://doc.akka.io/docs/akka/ 2.4.4/scala/persistence.html）。Eventuate 使用事件溯源的 actor 来处理命令，并用事件溯源视图和写程序处理查询（投影模型）。使用事件溯源处理器，可以在 Eventuate 中构建事件处理通道，这些通道可以构成分布式 CQRS 实现的主干。

本节是关于实现基于分布式 CQRS 领域服务的参考，并不会实现服务的具体细节。如果需要一个实现，可以查看诸如 Eventuate 之类的框架，但在大多数情况下，可能不需要分发式服务，哪怕模型中多个边界上下文在空间和时间上存在耦合。将数据存储保存到本地边界上下文，并确保每个边界上下文中的服务只访问存储中的数据。将每个边界上下文设计为独立的管理和部署单元，这种设计技巧有一个很流行的名字：*微服务*。Chris Richardson 提供了一系列很赞的模式，可以按照这些模式来设计基于微服务的架构（http://microservices.io/patterns/microservices.html）。

这个范式最重要的作用是边界上下文之间有了清晰的分离，而且不必管理跨上下文事件的因果关系。当然，也不是强制在所有边界上下文中实现 CQRS，每个上下文可以有其自己的数据管理技术。但由于上下文之间进行了解耦，就不必再处理跨上下文服务数据一致性的冲突。最终的结果是，在每个决定实现 CQRS 的边界上下文中，可以采用本章中讲到的纯函数式实现。

8.4.7 实现的总结

我们已经讲了一个很长的故事，也就是持久化聚合的事件溯源版本的完整实现。我们没有尝试使用通用框架，而是遵循一个特定的用例来观察模型的逐步演化。以下内容通过对总体实现的总结来全面回顾一下如何解决特定用例的问题：

1. 为事件定义一个代数。使用代数数据类型为每种事件类型定义特定的数据构造函数。
2. 将该命令定义为事件上的 free monad。这也使得命令可以组合，可以通过使用 for 表达式来构建组合命令。生成的命令仍然只是一个数据类型，没有任何相关联的语义。因此，命令是我们构造模式下的纯抽象。
3. 使用解释程序中的事件代数来执行命令处理。这样就可以在解释程序中处理所有作用，保证命令核心抽象的纯粹性。成功处理后，命令将事件作为解释逻辑的一部分进行发布，并附加到事件日志中。

4. 定义合适的读模型（也就是投影），并在写入端使用读模型处理事件流的适当的策略。

8.5 其他持久化模型

CQRS 和事件溯源提供了采用函数式实现的持久化模型。事件只是一系列函数，描述了已经发生的事情。事件溯源的许多实现提供了用于交互的函数式接口，这并不只是巧合。但这个范式在架构上有一定复杂性，这是我们不能忽略的。

- 不同的范式：它与数据层有完全不同的工作方式，与我们目前为止所使用的基于 RDBMS 的应用差了十万八千里。我们不能忽视这个事实，它不是大多数数据架构师所熟悉的领域。
- 操作的复杂性：使用单独的读 / 写模型，需要管理更多的移动部件。这也增加了架构操作的复杂性。
- 版本控制：使用长期领域模型，一定会遇到模型组件发生变化的情况。这将导致事件结构随时间变化，必须通过事件版本对此进行管理。有许多现成的方法，但没有一个是容易的。如果计划对应用程序采用事件溯源的数据模型，请从第一天起就考虑版本化的事件。

鉴于这些问题，可以在领域模型中对数据采用函数式关系型模型的方式，而不需要使用对象关系型框架来维持模型的可变性。有些库可以帮我们做这件事：借助不可变代数数据类型和函数式组合器的优势，与领域 API 一起与底层 RDBMS 进行交互。本节将简要介绍 Slick，一个开源的函数式关系型映射库。但不会详细讲述 Slick 如何开展工作。相关的详细信息，请参阅 Slick 网站（http://slick.typesafe.com）或 Richard Dallaway 和 Jonathan Ferguson 所著的 *Essential Slick*（http://underscore.io/books/essential-slick/）。

8.5.1 将聚合作为 ADT 映射到关系型表

接下来继续使用 `Account` 聚合的例子，清单 8.9 定义了客户帐户的聚合以及一个段时间内的所有余额。请注意，类型 `Balance` 包含一个字段 `asOnDate`，用于保存记录余额数额（`amount`）的日期。

清单 8.9 `Account` 和 `Balance` 聚合

```
import java.sql.Timestamp

case class Account(id: Option[Long],
  no: String,
```

```
  name: String,
  address: String,
  dateOfOpening: Timestamp,
  dateOfClosing: Option[Timestamp]
)

case class Balance(id: Option[Long],
  account: Long,
  asOnDate: Timestamp,
  amount: BigDecimal
)
```

注意，在 Balance 聚合中保存了对帐户 ID 的引用。

这是一个浅显的聚合设计，我们没有在 Balance 中存储对 Account 的引用，而是存储了一个帐户 ID。在处理底层关系型模型时，这通常是一个好的做法，因为它使领域模型聚合更贴近底层的关系型结构。在 Vaughn Vernon 所著的 *Effective Aggregate Design* 一书中就有一系列优秀文章，介绍了设计聚合的最佳实践（参见 https://vaughnvernon.co/?p=838）。

使用 Slick，可以将聚合结构映射到底层关系型模式。清单 8.10 展示了如何完成此操作以及如何将表结构定义为内部的 Scala DSL。

清单 8.10 使用 Slick 将表定义为 Scala DSL

```
import Accounts._
import Balances._
class Accounts(tag: Tag) extends Table[Account](tag, "accounts") {
  def id = column[Long]("id", O.PrimaryKey, O.AutoInc)
  def no = column[String]("no")
  def name = column[String]("name")
  def address = column[String]("address")
  def dateOfOpening = column[Timestamp]("date_of_opening")
  def dateOfClosing = column[Option[Timestamp]]("date_of_closing")

  def * = (id.?, no, name, address, dateOfOpening, dateOfClosing)
    <> (Account.tupled, Account.unapply)
  def noIdx = index("idx_no", no, unique = true)
}
object Accounts {
  val accounts = TableQuery[Accounts]
}
class Balances(tag: Tag) extends Table[Balance](tag, "balances") {
  def id = column[Long]("id", O.PrimaryKey, O.AutoInc)
  def account = column[Long]("account")
  def asOnDate = column[Timestamp]("as_on_date")
  def amount = column[BigDecimal]("amount")

  def * = (id.?, account, asOnDate, amount)
    <> (Balance.tupled, Balance.unapply)
  def accountfk = foreignKey("ACCOUNT_FK", account, accounts)
    ( .id, onUpdate=ForeignKeyAction.Cascade,
```

请注意如何将 ADT Account 映射到表定义

将帐户主键定义为一个自动增量的字段

定义映射到 Account ADT 的查询投影

```
    onDelete=ForeignKeyAction.Cascade)
}
object Balances {
  val balances = TableQuery[Balances]
}
```

定义链接 Account 和 Balance 的外键

将 ADT Account 和 Balance 映射到底层表结构，并将它们定义成能被底层数据库操作。Slick 通过其核心内部精心设计的结构来完成这种映射。我们获得的是数据库结构和类型的不可变代数之间无缝的互操作性。接下来在讨论如何使用 Slick 的函数式组合器来处理数据时，将会看到如何操作。

8.5.2 操作数据（函数式）

作为 Scala 的库，Slick 提供了用于操作数据的函数式组合器，看起来很像 Scala Collections API。这使得 Slick API 对于 Scala 用户来说也更加容易使用。考虑清单 8.11 中的例子，我们希望使用其中定义的模式对数据库进行查询，查询指定时间间隔内特定 Account 的 Balance 记录集合。

清单 8.11 使用 Slick 的函数式组合器查询 Balance

db 是数据库句柄，并且在 fromDate 到 toDate 的日期范围内查询 accountNo 的余额。

```
def getBalance(db: Database, accountNo: String, fromDate: Timestamp,
  toDate: Timestamp) = {
  val action = for {
    a <- accounts.filter(_.no === accountNo)
    b <- balances if a.id === b.account &&
                     b.asOnDate >= fromDate && b.asOnDate <= toDate
  } yield b
  db.run(action.result.asTry)
}
```

db.run 返回一个 Future，可以以非阻塞的方式进行组合。

熟悉的表达式语法，就像在 Scala 集合中一样，只不过这里是基于一个底层数据库。

可以看到，核心查询 API 类似于 Scala 中集合的模型。查询 accounts.filter 从 Accounts 表中获取一个投影，并通过 for 表达式与 Balances 表上的查询进行绑定。这实际上是单子化实现的关系型连接。有关联接和查询的更多变体，请参阅 Slick 的相关文档（http：// slick.typesafe.com）。查询返回一个 Future，也正是使模型具备响应性的重要核心之一。我们可以与其他返回 Future 的 API 进行组合，构建更大的非阻塞 API，而不是在 Future 上等待。不过，通过允许将数据库查询结果直接流入到基于 Akka Streams 通道的方式，Slick 提供了具有更好响应性的 API。接下来将简单地看下这个功能。

8.5.3 到 Akka Streams 管道的响应式获取

考虑这样一个用例：从数据库中获取大量数据并将其作为领域逻辑的一部分进行处理。为了响应边界延迟并保证服务器的响应时间，可以借助流的力量。Slick 等响应式框架允许将数据库查询结果作为源直接发布到 Akka Streams 通道中（http://doc.akka.io/docs/akka/ 2.4.4 / scala / stream / index.html）。

这里有一个简单的查询，它获取特定时间段内按帐户分组的所有余额的总和。这对于任何服务于零售客户的金融公司的数据库来说必然是一个巨大的结果集。清单 8.12 展示了如何用 Slick 实现查询。

清单 8.12 生成结果流的余额查询

返回一个 DatabasePublisher，它向响应式流进行发布

```
type BalanceRecord = (String, Timestamp, Timestamp, Option[BigDecimal])
def getTotalBalanceByAccount(db: Database, fromDate: Timestamp,
  toDate: Timestamp): DatabasePublisher[BalanceRecord] = {
  val action = (for {
    a <- accounts
    b <- balances if a.id === b.account && b.asOnDate >= fromDate
                  && b.asOnDate <= toDate
  } yield (a.no, b)).groupBy(_._1).map { case (no, bs) =>
    (no, fromDate, toDate, bs.map(_._2.amount).sum)
  }
  db.stream(action.result)
}
```

流 API，用于返回数据流

留意如何在使用 Scala DSL 的查询中实现 groupBy

这个清单包含一些以响应式的方式访问数据库的重要结论。

- *后端查询能力*：Slick 提供了大量组合器在服务器级别处理数据，并且可以为它们生成优化的 SQL。在清单 8.12 中，`groupBy` 的使用就是一个类似的案例。通过使用 `groupBy`，可以仅在服务器级别进行分组（与使用 SQL 一样），这样就可以节省昂贵的数据传输，在客户端执行格式化。
- *响应式流集成*：查询返回一个 `DatabasePublisher`，我们可以直接将它作为 `Source` 挂到 Akka Streams 流图上，例如，`val accountSource:Source[BalanceRecord] = Source(getTotalBalanceByAccount(..))`。

总而言之，可以将关系型数据库和一个适当的函数式库一起使用，将数据库建模为响应式持久层。

8.6 总结

本章介绍了在底层数据库中的持久化领域模型。如果说用关系型数据库来存储领域模型元素的时代已经过去，新的范式已经出现，那么我们已经学会更好地将函数式领域模型与底层存储相匹配。事件溯源和 CQRS 避免了数据就地更新的概念，直接将领域模型存储为不可变的事件流。本章的主要结论如下。

- CRUD 不是持久化的唯一模型：本章讨论了使用事件作为真相来源的原理，并且引入事件溯源作为一种存储领域事件的技术，这使得模型具有更好的可审计性和可追踪性，同时在感知底层存储的业务价值上有了巨大的提升。它不仅是关系型数据的存储，也是领域事件的存储。
- 实现函数式事件溯源领域模型：本章演示了如何使用函数式技术实现事件溯源模型。现在有很多其他的可用实现，但是我们采用了与函数式编程的概念非常吻合的方法。free monad 仅提供了一个纯粹的命令和事件代数的实现，副作用则由解释程序来处理。
- FRM，而不是 ORM：如果决定使用 CRUD 模型进行持久化（并且有正当理由这样做），请使用函数式关系型映射框架，比如 Slick。本章简要讨论了如何通过一个对应底层关系型数据库的纯函数式接口来完成此项内容。

9 测试领域模型

本章包括

- 设计可测试领域模型
- 学习为什么基于 xUnit 的测试是不充分的
- 引入基于属性的测试
- 将属性用作领域行为的可执行蓝图

本章涵盖了如何测试领域模型。开头会解释可测试模型的含义以及可测试性与模型架构的模块化如何相关联。接着会从前面章节讨论的模型组件中选取一些例子，研究如何对它们进行测试。然后简要讨论基于 xUnit 测试的缺点，并深入讨论使用 ScalaCheck 开展基于属性测试的细节。

图 9.1 展示了本章的内容。

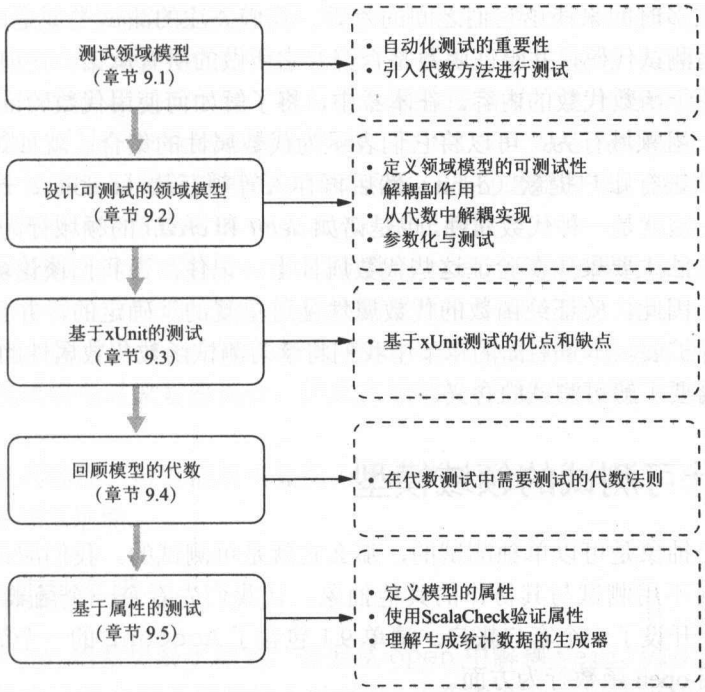


图 9.1 本章学习过程。

9.1 测试领域模型概述

在深入领域模型测试之前，首先需要了解本书所说的可测试性究竟是什么意思。毕竟，我们可能有一组团队成员用大量的测试数据对领域模型进行手工测试，但这不是本书所说的测试定义。手工测试的主要问题是它不可重复。在对领域模型实现进行更改后，如何确保它仍按照规范工作？我们会雇用同一组测试团队成员并重复整个测试过程吗？如何找出模型的哪些部分受到了昨天晚上所做更改的影响？有时候，手工测试是靠不住的。

我们需要自动化测试，因为：

- 可重复
- 可以使用附加数据集进行扩展
- 可维护
- 额外功能扩展
- 能够与构建系统集成

测试可以对应到各种粒度级别。例如，可以包括单元测试、模块测试和集成测试，

本章不会花太多时间来谈论它们之间的差异。我们关注的测试类型是白盒测试：在函数级别编写测试代码，并验证函数是否符合它所做的所有承诺。在前面的章节中，讨论了很多关于函数代数的内容。在本章中，将了解如何使用代数验证函数的属性。函数体现了一组领域行为，可以将它们表示为代数属性的集合。就如银行领域中的一个例子：从银行账户提款（debit）然后再存入等额存款（credit），这将恢复账户的原始余额。这就是一种代数属性，也是诸如 *debit* 和 *credit* 的领域行为必须遵守的。我们的测试方法主要聚焦在验证这些代数属性上。记住，当我们谈论函数时，谈论的是纯函数。因此，验证纯函数的代数属性是可重复的、确定的，并且可以使用附加数据集进行扩展。本章后面的章节中我们将学习测试函数代数属性的相关技术。

但首先需要了解可测试的含义。

9.2 设计可测试的领域模型

如果一个抽象是可以单独测试的，那么它就是可测试的。我们应该能够直接测试一个抽象而不用测试与其协作的其他抽象。让我们先看个一个函数实现的例子，该函数在银行开设了一个客户帐户。清单 9.1 包含了 Account 的一个简单实现，我们会重点介绍 open 函数行为方面。

清单 9.1 面向可测试的领域模型

```
case class Account(no: String, name: String, idNo: String,
  dateOpened: DateTime, dateClosed: Option[DateTime])

trait AccountService {
  type Error = String
  type ErrorOr[A] = Error \/

  def open(no: String, name: String, idNo: String,
    dateOpened: DateTime): ErrorOr[Account] = {
    val isValid: Boolean = verifyId(idNo, name)
    if (isValid) {
      //.. other validations
      Account(no, name, idNo, dateOpened, None).right
    } else s"Id ($idNo) validation failed".left
  }

  def verifyId(idNo: String, name: String): Boolean = {
    // possibly an expensive call that needs to communicate
    // with external systems, maybe over a Web service
    // stubbed here
    true
  }
}
```

Account 代数数据类型

open 要么返回错误，要么返回开设的账户

外部服务调用——一个副作用

open 函数是不是很容易测试？开始时，它调用函数 verifyId 来进行客户 ID 和名称的验证。虽然已经剔除了 verifyId 的实现，但实际上，这里可能需要通过 Web 服务与第三方实现进行交互，然后再返回验证结果。这个调用会在 open 函数里引发副作用，而且在纯函数式编程的词汇表中这被认为是不纯粹的。诚然，我们很难测试这种包含外部服务调用的 open 函数，但更重要的是，测试 open 函数正确性的测试用例不应该负责测试 verifyId 的正确性。

我们需要找到一个方式将 verifyId 移出 open 函数的测试路径。一个常见的做法是使用一个模拟库，通过提供模拟实例来模拟外部依赖，并允许开发人员验证被测试的系统。除了以另一个库的形式引入外部依赖之外，还引入了大量的模板。它们通常与领域模型对象紧密耦合，因此在领域模型更改的情况下它们也需要被重新设计。

通过函数式编程，可以借用函数的力量来模拟实现，也就消除了在开发中引入另一个外部库的复杂性。

9.2.1 解耦副作用

为了使 open 函数更易于测试，需要从 open 中解耦 verifyId 的副作用。本书之前的章节中已经介绍了依赖注入的技巧。[1] 在本章节中将使用相同的技巧：使 verifyId 成为可以从外部注入的抽象的组成部分。一旦使 verifyId 成为环境的一部分，就可以灵活地注入一个绕过外部服务调用整个逻辑的纯实例。本书之前谈到的往领域服务中注入仓储，第一步就是让 open 返回一个函数。清单 9.2 就做了这件事。

清单 9.2 面向可测试的领域模型

```
case class Account(no: String, name: String, idNo: String,
  dateOpened: DateTime, dateClosed: Option[DateTime])

trait IdVerifier {
  def verifyId(idNo: String, name: String): Boolean
}

trait AccountService {
  type Error = String
  type ErrorOr[A] = Error \/

  def open(no: String, name: String, idNo: String,
    dateOpened: DateTime)
    : IdVerifier => ErrorOr[Account] = { (v: IdVerifier) =>
    if (v.verifyId(idNo, name)) {
      //.. other validations
      Account(no, name, idNo, dateOpened, None).right
```

提供 verifyId 方法的 trait

现在 open 返回一个接受 IdVerifier 的函数

1 在3.3.6节中，我们讨论了如何向服务中注入存储。如果需要，可以回顾下那个章节。

```
    } else s"Id ($idNo) validation failed".left
  }
}

case class MockIdVerifier() extends IdVerifier {
  def verifyId(idNo: String, name: String) = true
}
```

❶ verifyId 的模拟实现以进行测试

这里，open 返回一个接受参数 IdVerifier 的函数，它是提供 verifyId 代数的模块。我们可以有多个模块实现：一个用于生产系统，可以与外部服务调用交互，而另一个是用于测试的模拟实现 ❶。总而言之，我们已经通过将该函数从一个非纯函数的实现中解耦，从而使其更易于测试。

9.2.2 为领域函数提供自定义解释程序

我们刚刚看到通过单个函数的净化来增强可测试性的实例。可以再进一步提升抽象级别，并为整个代数提供定制的纯实现。第 5 章中讨论 free monad 时，就已经学过了这一技术。让我们在领域模型测试的背景下再次讨论这个主题。

在 5.5 节的清单 5.6 中，我们为帐户存储设计了一个代数，用于处理帐户与基础数据层的交互。以下是我们定义的代数：

```
sealed trait AccountRepoF[+A]
case class Query(no: String) extends AccountRepoF[Account]
case class Store(account: Account) extends AccountRepoF[Unit]
case class Delete(no: String) extends AccountRepoF[Unit]
```

动作的基本 trait

将每个动作表示为 ADT（纯数据）

并且基于 trait AccountRepoF 定义了一个 free monad：

```
type AccountRepo[A] = Free[AccountRepoF, A]
```

Free 给我们的是一个代数 monad。现在可以使用代数的各种元素，并对它们进行组合以构建更大的抽象。例如，可以组合 Query 和 Store 来实现更新功能。[1] 代数是一个纯抽象，它没有 Query、Store 和 Delete 的任何语义，它是实现这些语义的代数解释程序。因此，设计可测试 API 的技术是为了使 API 仅依赖于代数，同时具有来自环境且可插入的解释程序（正如在 9.2.1 节中为 IdVerifier 所做的）。

1 参见清单5.8提供的相关例子。

在生产系统中，通常使用企业级数据库来实现存储。但是对于测试，应该能够用一个更简单的替代实现来测试API，例如Map [K, V]。一个基于free monad的实现提供了完整的工具。我们可以有独立的解释程序：一个是基于企业级数据库实现语义的生产应用，一个是用于使用底层Map作为存储的数据结构的测试。第5章介绍了这一技术，并在清单5.8中基于Map为AccountRepository实现了一个解释程序。

以下是5.5节中实现的一个例子，可以使用代数在客户帐户上定义一个组合动作序列。这仅仅是组合在一起的纯数据类型，没有任何与open、credit、debit或者query明确关联的行为：[1]

```
val composite = for {
  x <- open("a-123", "debasish ghosh", Some(today))
  _ <- credit(x.no, 10000)
  _ <- credit(x.no, 30000)
  _ <- debit(x.no, 23000)
  a <- query(x.no)
} yield a
```

现在已经为组合操作提供了API，可以使用解释程序作为计算的最后一步。解释程序接受代数，然后通过剥离free monad的层并解释内部functor的代数来注入语义。可以使用基于Map的解释程序进行测试，AccountRepoMapInterpreter().apply(composite)，或者在生产中使用基于数据库的解释程序——例如，AccountRepoOracleInterpreter().apply(composite)。总而言之，基于free monad的实现为API可测试提供了极好的支持。

9.2.3 实现参数化与测试

使领域模型更容易测试的一个方法是不再为一部分代码编写测试代码：可以让其他人来为我们做这件事。本节将介绍如何使编译器为模型做一些测试，从而减少必须要编写的测试代码数量。这不是什么新的概念。在本书前面章节中应该已经介绍过这种技术，4.3节中还详细讨论过。它被称为参数化：参数化的函数是在一种或多种类型上的多态，并且仅依据这些类型所拥有的代数来实现。举个例子，如果有一个函数def accumulate [A:Monoid](a1:A,a2:A)，函数的实现将仅仅是Monoid的代数。在这种情况下，可以在参数化为Monoid的函数上调用accumulate。

本书在第4章中讨论monoid时，已经介绍了如何通过抽象领域行为和领域上下文来实现参数化的领域模型。用泛型代码替换特定类型代码可以带来更好的抽象

1 所有这些都是在5.5节中用存储的代数所实现的操作。查阅5.5节了解更多细节。

性，也因此在不同的上下文中具有更好的可重用性。在4.1.2节中实现了一个泛型组合器mapReduce，如下所示：

```
def mapReduce[F[_], A, B](as: F[A])(f: A => B)
  (implicit fd: Foldable[F], m: Monoid[B]) = fd.foldMap(as)(f)
```

请注意我们是如何用f:A=>B抽象一个操作的，并通过类型构造器F应用操作到上下文上。在领域模型的上下文中，f可以是任何领域行为，而F则是从数据库中获取的领域元素的集合。

这些参数化函数的主要优点是可以使用满足代数的任何类型来测试它们，而不需要是来自领域的复杂类型。在前面的例子中，可以使用f的函数和F[A]的整型列表来测试mapReduce。更重要的是，可以为参数化函数定义泛型代数属性，并使用它们来验证正确性，这使得它们可以用满足代数的任何类型来测试。编译器确保传递正确的类型，而代数属性确保它正确工作。想象一下我们可以少写多少测试代码，而要做的只是用一小段参数化的泛型代码替换特定领域类型和操作的所有实例。

9.3 基于xUnit的测试

现在测试模型最常用的方法是使用基于xUnit的测试方法。JUnit（www.junit.org）和NUnit（www.nunit.org）等测试框架提供了相关抽象来单独测试模型中的单元。我们可以定义自己的单元（比如，对于基于类的面向对象方法，通常将一个类定义为一个单元），然后，编写断言，根据提供的各种数据集来验证类的不同方法。

在确定要测试的单元之后，在编写基于xUnit的测试代码时要遵循以下主要步骤。

1. 识别出要测试的方法（函数）。
2. 从方法中识别出场景。例如，可能想要为方法的快乐路径编写一个测试场景：所有验证都通过，并且从方法中得到一个成功的返回。
3. 提供一个手工数据集来测试场景。
4. 检查断言。

这种方法在第三步中存在不足，因为会使用1~2个事先准备好的数据来测试场景。通常这是不充分的，因为从商业角度来说手动准备的数据集从来都是无穷无尽的。通常我们会错过在生产环境中可能出现的边缘情况或边界条件。在下一节中将看到如何通过使用基于属性的测试解决这个问题。

9.4 回顾模型的代数

在第 3 章中已经说明模型的代数由以下 3 部分组成：

- 类型，用来建模领域实体、值对象等。
- 操作类型的函数，相关函数组合在一起成为模块。
- 法则或业务规则。

在前面的章节中，已经看到了前两点如何作为领域模型的一部分进行展现。我们讨论了各种方法，如何设计类型领域模型以及将领域行为分解为对类型操作的函数。现在将讨论一下领域代数的第三个组成部分：通过业务规则的代数验证法则。注意，这些是对业务规则，而不仅仅是对文档的验证。我们编写的所有属性都将证明统治着领域模型代数的业务规则的正确性，这些属性形成了一个和类型、函数同等重要的模型组件。因此，在设计属性集时请小心对待它们。图 9.2 给出了这 3 个组件如何构成领域代数的示意图。下一节将讨论将属性作为领域行为验证工具的所有实现细节。

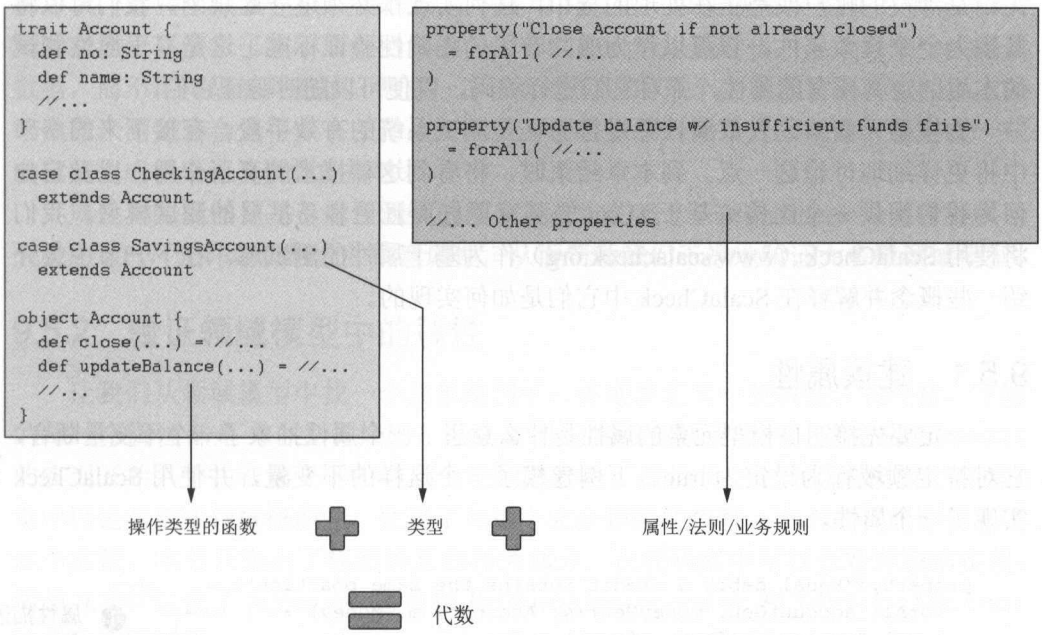


图 9.2 完整的领域代数包括 3 个组件的统一：类型、操作类型的函数和业务规则。

9.5 基于属性的测试

到目前为止，本书的核心主题之一就是领域建模的代数。本书会把一个模型想象成代数的联合，每一个代数集合都有独立的边界上下文描述。在边界上下文中，特定类型对特定领域概念进行建模。我们已经看到了一些类型，比如 Account、Customer、Balance 等，它们根据领域通用语言的相应实体进行命名。每种类型都参与领域行为的建模，我们将紧密相关的行为分组到单独模块中。每个模块都是函数的集合，它们对类型进行操作并对特定代数进行建模。这些类型不会孤立地挂起，它们建模的代数通常会被组合成其他代数。在第 4 章中，介绍了如何通过将领域特定类型的代数如 Trade、Order、Account 和 Market 与泛型代数结构包括 Monad 和 Kleisli 组合到一起，得到一个交易生成算法的实现。

当组合代数时，也会得到代数的行为。可以将这些行为视为具有基于领域规则的代数属性。举个例子，为两个客户帐户之间的资金转账进行建模时，领域规则表示从一个帐户扣除的金额必须等于存入到另一个帐户的金额。这是一个领域规则，无论是涉及的帐户或是正在处理的货币，任何实现都必须遵守此规则。我们可以将其视为一个代数属性，它可以作为成功转账的正确性验证标准。这是基于属性测试的本质：定义模型的属性，并对它们进行编码，以便可以随时验证它们。

为模型元素编码代数属性并进行验证是测试系统的有效手段。在接下来的小节中将更详细地讨论这一点。到本章结束时，将看到这种技术的真正作用，以及它如何为我们提供一个比传统基于 xUnit 的测试更好而且更容易扩展的测试模型。我们将使用 ScalaCheck（www.scalacheck.org）作为基于属性的测试库。在下一节中会介绍一些概念并解释在 ScalaCheck 中它们是如何实现的。

9.5.1 建模属性

一定要先搞明白模型元素的属性是什么意思：一个属性抽象了一个不变量断言，它对特定领域行为设定为 true。下例建模了一个这样的不变量，并使用 ScalaCheck 实现了一个属性。

```
property("Equal debit & credit retains the same position") =
  forAll(accountGen, moneyGen)(a: Account, m: Money) => {
    val Success((before, after)) = for {
      p <- position(a)
      r <- credit(p, m)
      q <- debit(r, m)
    } yield (p, q)

    before == after
  })
```

❶ 属性描述

forAll 将创建属性

断言验证

我们没有定义过任何在前面代码中所使用的领域类型或函数（如 position、debit 和 credit）。但它们很直观，可以很容易弄清它们在个人银行领域的上下文中做了什么。[1] 让我们花一些时间通过剖析每一段有趣的代码来解释用 ScalaCheck 实现属性的整体架构。详细的例子可以在本书的后续部分以及代码库中找到。

整个代码片段显示了如何定义属性。在 ❶ 中描述了属性所要验证的断言。无论处理何种类型的帐户，相同金额的 debit 和 credit 必须保持帐户初始位置（position）不变。[2] 我们还可以将此陈述视为描述编码业务规则的属性。

forAll 是根据一个断言以及一个或多个生成器创建属性的函数，生成器生成用于验证属性的数据。在 ScalaCheck 中生成器被建模为 Gen[A]，并且是生成类型 A 实例的抽象的基础框架。在最简单的情况下，forAll 使用一个 Gen[A] 和一个断言 A => Boolean，然后检验断言的可满足性。ScalaCheck 自带一套内置类型的生成器。我们也可以实现自己的生成器，并为它们提供 forAll 来创建与领域模型相关的自定义属性。在前面的示例中，属性定义使用了 accountGen 和 moneyGen 这两个生成器，它们分别生成 Account 实例和 Money 实例，这两个抽象来自我们的领域，这些生成器生成我们在断言中使用的数据，以实现需要验证的领域规则。同时在这里还演示了基于属性测试的另一个功能：可以通过传入生成器来生成测试数据，而不用再硬编码数据。想象一下，如果可以用几个生成器替换所有的手工编码数据，测试套会变得多么干净。在下一节中将更深入地研究生成器，并定义一些自己的生成器来定义模型的属性。

属性和生成器是基于属性测试的两个最基本的概念。ScalaCheck 为两者提供了大量的 API，允许逐步组合并构建属性，从而帮助我们干净地验证大多数领域模型。

9.5.2 验证领域模型中的属性

让我们从领域模型中找一个抽象的例子，并逐步定义一些属性。请注意，下面的每一个属性都是一个业务规则，也是模型实现所必须遵守的。首先从 Account 的实现开始入手。在第 3 章中谈到智能构造器时，提供了一个类似的模型。在第 6 章中讨论响应式领域模型时，使用了与这个完全相同的模型。本书的代码库包含了这个实现，本节只突出了模型的某些相关部分，在代码库中可以查看详细的实现。清单 9.3 中包含了 Account 抽象以及 CheckingAccount 和 SavingsAccount 的代数数据类型。

---

1 你一定已经看过书中定义的所有这些函数。

2 这里的位置也就是余额。简单起见，我们假设这是一个单线程的模型，而且在debit和credit之间没有发生其他事。

清单 9.3 Account 抽象

```
import java.util.{ Date, Calendar }

object common {
  type Amount = BigDecimal
  def today = Calendar.getInstance.getTime
}

import common._

case class Balance(amount: Amount = 0)

sealed trait Account {                          <-- Account 的基本 trait
  def no: String
  def name: String
  def dateOfOpen: Option[Date]
  def dateOfClose: Option[Date]
  def balance: Balance
}

final case class CheckingAccount (no: String, name: String,
  dateOfOpen: Option[Date], dateOfClose: Option[Date] = None,
  balance: Balance = Balance()) extends Account                 <-- 支票账户和存储账户的 ADT

final case class SavingsAccount (no: String, name: String,
  rateOfInterest: Amount,
  dateOfOpen: Option[Date], dateOfClose: Option[Date] = None,
  balance: Balance = Balance()) extends Account
```

接着，在清单 9.4 中定义了一些智能构造器，允许创建 CheckingAccount 和 SavingsAccount 的有效实例。第 3 章中已经涵盖了智能构造器，它们提供了用于创建领域对象有效实例的接入点。

清单 9.4 Account 的智能构造器

```
object Account {
  def checkingAccount(no: String, name: String, openDate: Option[Date],
    closeDate: Option[Date],
    balance: Balance): NonEmptyList[AccountException]\/ Account = {

    val od = openDate.getOrElse(today)

    (
      validateAccountNo(no) |@|
      validateOpenCloseDate(openDate.getOrElse(today), closeDate)
    ) { (n, d) =>
      CheckingAccount(n, name, d._1, d._2, balance)
    }.disjunction                                   <-- 智能构造器确保在创建时获得有效的实例
  }

  def savingsAccount(no: String, name: String, rate: BigDecimal,
    openDate: Option[Date], closeDate: Option[Date],
    balance: Balance): NonEmptyList[AccountException] \/ Account] = {
```

```
    val od = openDate.getOrElse(today)

    (
      validateAccountNo(no) |@|
      validateOpenCloseDate(openDate.getOrElse(today), closeDate) |@|
      validateRate(rate)
    ) { (n, d, r) =>
      SavingsAccount(n, name, r, d._1, d._2, balance)
    }.disjunction
  }
}
```

智能构造器确保在创建时获得有效的实例

这段代码非常清晰，因为现在我们一定已经很熟悉智能构造器的所有习惯用法。每个智能构造器内部都包含相当数量的验证，以确保在成功构建后可以获得 CheckingAccount 和 SavingsAccount 的有效实例。通过构建器将验证拼接在一起，这样所有的验证错误都会积累在函数返回类型中。最后，清单 9.5 呈现了 Account 需要实现的一些领域功能，以便在模型中执行我们所期望的行为。这些为定义属性提供了基础。简洁起见，此清单隐藏了所有验证函数，详情可以查看代码库中的实现。

清单 9.5 Account 的领域功能

```
object Account {
  def close(a: Account, closeDate: Date)
    : NonEmptyList[AccountException] \/ Account = {
    (validateAccountAlreadyClosed(a) |@|
     validateCloseDate(a, closeDate)) { (acc, d) =>
      acc match {
        case c: CheckingAccount => c.copy(dateOfClose = Some(closeDate))
        case s: SavingsAccount  => s.copy(dateOfClose = Some(closeDate))
      }
    }.disjunction
  }

  def updateBalance(a: Account, amount: Amount)
    : NonEmptyList[AccountException] \/ Account = {
    (validateAccountAlreadyClosed(a) |@|
     checkBalance(a, amount)) { (_, _) =>
      a match {
        case c: CheckingAccount =>
          c.copy(balance = Balance(c.balance.amount + amount))
        case s: SavingsAccount  =>
          s.copy(balance = Balance(s.balance.amount + amount))
      }
    }.disjunction
  }
}
```

验证帐户关闭行为

请考虑一个业务规则：如果有一个客户帐户尚未被关闭，那么我们应该能够关闭该账户，并有一个有效的关闭日期。让我们尝试编写一个属性来验证这个业务规则。

清单 9.6 用代数方法验证帐户关闭行为

```
property("Close Account if not already closed") =
  forAll(validCheckingAccountGen) {
    _.map { account =>
      account.dateOfClose.map(_ => true).getOrElse(
        close(account,
              account.dateOfOpen.map(aDateAfter(_)).getOrElse(common.today)
        ).isRight == true
      )
    }.getOrElse(false)
  }
```

forAll 生成的属性是通用量化属性。给定一个生成器，forAll 为生成器生成的每个值创建属性。在清单 9.6 中，将为生成器 validCheckingAccountGen 生成的每个帐户创建属性。我们不必为创建属性而手工传入 Account 的值。由于组成实现的生成器的基础是参数化的（类型 A）Gen[A]，所以所有属性都是通用量化的。注意清单 9.6 里 forAll 的断言，它取得一个生成器生成的帐户，如果帐户已经关闭则返回 true。否则，它将调用 close 函数并设定有效关闭日期，如果 close 成功则返回 true。

这个例子的主要目的是告诉我们可以用代数推导以完全声明的方式来验证领域行为。清单 9.6 实现的属性等同于由某个生成器生成的 *accounts*（*a*）代数表达式，关闭尚未关闭的帐户必须成功。

用更新余额验证失败

在进一步探索生成器之前，让我们通过使用属性的代数推导来验证另一个业务规则。清单 9.5 包含一个领域函数 updateBalance，它被用来更新帐户的余额。此更新既可以是提取资金，也可以是存入资金。如果尝试从没有足够余额的帐户中进行扣款，那么交易一定会失败。而且如果尝试对已关闭的帐户执行相同的操作，则操作也应该失败。在清单 9.5 的 updateBalance 函数中，这两个验证都有被执行。让我们检验一下验证工作在清单 9.7 中是否正常工作。

清单 9.7 用代数方法验证更新余额行为

```
property("Update balance on closed account fails") =
  forAll(validClosedCheckingAccountGen, genAmount) { (creation, amount) =>
    creation.map { account =>
```

```
      updateBalance(account, amount) match {
        case -\/(NonEmptyList(AlreadyClosed(_))) => true
        case _ => false
      }
    }.getOrElse(false)
  }

property("Update balance on account with insufficient funds fails") =
  forAll(validZeroBalanceCheckingAccountGen, genAmount) {
    (creation, amount) =>
      creation.map { account =>
        updateBalance(account, -amount) match {
          case -\/(NonEmptyList(InsufficientBalance(_))) => true
          case _ => false
        }
      }.getOrElse(false)
  }
```

这里，我们使用生成器生成适当类型的数据。在第一个例子中，生成器生成了关闭的帐户。在下一节中将会看到它是如何做到的。在清单 9.7 的第二个例子中，生成器生成了余额为零的帐户。在第一种情况下，当 updateBalance 失败并抛出一个 AlreadyClosed 异常时，我们宣布属性已验证。在第二种情况下，根据正确的业务规则，将失败并抛出异常 InsufficientBalance。在这里就不提供额外的实现细节了，可以从本书的代码库中获取它们。

这个例子的重点类似于我们讨论的关闭帐户——通过代数属性验证业务规则。通过这种方式，可以使用代数方法为整个模型编码业务规则，并使模型可验证而且更加稳健。

通过类型还是代数属性做验证

本书的一个重要主题就是使用类型进行建模的重要性与日俱增。当我们使用一种语言时，如 Scala，它提供了一个强类型系统，那我们为什么不利用它的优势呢？类型提供了一种很好的编码领域逻辑的方法，在本书中已经看到过很多次。当使用类型编码逻辑时，不必再编写测试代码，编译器会自动执行。

那么为什么还需要代数属性来验证领域行为呢？

类型并不能充分表达每个领域的约束或规则。我们不可能只用类型来表达在清单 9.6 和 9.7 的上下文中讨论的规则，而需要对其验证进行明确的测试。基于属性的测试的优点是它允许我们生成通用的量化属性，因此测试过程跟通过手工编写数据集进行测试相比会更加地详尽。

重点是我们两个都需要。尽可能地使用类型，同时，以代数方式编码属性来验证业务规则。

9.5.3 数据生成器

属性和生成器是基于属性测试技术的左膀右臂。通用量化属性需要数据生成器来开展全面测试。两者的组合使得验证过程比基于 xUnit 的测试更加彻底，也更容易扩展。

在 ScalaCheck 中，生成器来自于 `org.scalacheck.Gen` 类。可以把 `Gen[A]` 看作 `Gen.Params=>Option[A]` 的函数。但 `Gen` 类提供了足够的组合器，并且提供了许多有用的方法来生成数据。可以在 www.scalacheck.org 上的 ScalaCheck 文档中查看完整的组合器列表。为了了解 `Gen` 的某些功能以及如何使用 for 表达式来组合它们并生成更复杂的数据，这里提供了一个通过 Scala Repl 的会话例子：

```
scala> import org.scalacheck._
import org.scalacheck._

scala> import Prop.forAll
import Prop.forAll

// generate an integer between 10 and 20
scala> Gen.choose(10, 20)
res0: org.scalacheck.Gen[Int] = org.scalacheck.Gen$$anon$3@3bb613d7

// generate samples from the above generators
scala> res0.sample
res1: Option[Int] = Some(11)

scala> res0.sample
res2: Option[Int] = Some(17)

scala> res0.sample
res3: Option[Int] = Some(16)
// compose generators using for comprehension
scala> for {
     |   a <- Gen.choose(10, 20)
     |   b <- Gen.choose(100, 200)
     | } yield (a, b)
res4: org.scalacheck.Gen[(Int, Int)] = org.scalacheck.Gen$$anon$6@77146bb6

scala> res4.sample
res5: Option[(Int, Int)] = Some((19,119))

scala> val vowel = Gen.oneOf('A', 'E', 'I', 'O', 'U', 'Y')
vowel: org.scalacheck.Gen[Char] = org.scalacheck.Gen$$anon$3@6f8751cb

scala> vowel.sample
res6: Option[Char] = Some(O)

scala> vowel.sample
res7: Option[Char] = Some(U)

scala> case class Balance(amount: BigDecimal)
defined class Balance
```

```
scala> val genAmount = for {
     |     value <- Gen.chooseNum(100, 10000000)
     |     valueDecimal = BigDecimal.valueOf(value)
     |   } yield valueDecimal / 100
genAmount: org.scalacheck.Gen[scala.math.BigDecimal] = …

scala> val genBalance = genAmount map Balance
genBalance: org.scalacheck.Gen[Balance] = …

scala> genBalance.sample
res8: Option[Balance] = Some(Balance(1))

scala> genBalance.sample
res9: Option[Balance] = Some(Balance(76918.86))

scala> genBalance.sample
res13: Option[Balance] = Some(Balance(14665.5))
```

为了进一步了解 ScalaCheck 中生成器是如何工作的，我们找了一个来自清单 9.6 的例子。validCheckingAccountGen 生成有效的 CheckingAccount 实例，于是我们就可以创建验证账户关闭行为的属性。清单 9.8 实现了 validCheckingAccountGen。

清单 9.8 CheckingAccount 数据生成器

```
import java.util.Date
import org.scalacheck._
import Prop.forAll
import Gen._
import Arbitrary.arbitrary

val amountGen = for {
  value <- Gen.chooseNum(100, 10000000)
  valueDecimal = BigDecimal.valueOf(value)
} yield valueDecimal / 100

val balanceGen = amountGen map Balance

implicit val arbitraryBalance: Arbitrary[Balance] = Arbitrary {balanceGen}

val validAccountNoGen = Gen.choose(100000, 999999).map(_.toString)

val nameGen = Gen.oneOf("john", "david", "mary")

def optionalValidCloseDateGen(seed: Date) =
  Gen.frequency(
    (8, Some(aDateAfter(seed))),
    (1, None)
  )
```

生成一个 BigDecimal 类型的 Account

生成 Account 的 Balance

生成有效的账户编号

生成有效的姓名

这里允许对 Balance 生成使用“任意”的组合器 ❶

❸ 生成有效的账户关闭日期

```
val validCheckingAccountGen = for {
  no <- validAccountNoGen
  nm <- nameGen
  od <- arbitrary[Date]
  cd <- optionalValidCloseDateGen(od)
  bl <- arbitrary[Balance]
} yield checkingAccount(no, nm, Some(od), cd, bl)
```

❷ 通过智能构造器生成支票账户

虽然有一些组合器我们还不是很熟悉，但可以从此清单中大致了解支票账户生成器是如何工作的。让我们来看一些特殊功能，它们被用来为一个领域类实现生成器，如 CheckingAccount。

- 生成余额：gountGen 和 balanceGen 都非常浅显易懂，使用 Gen 的标准组合器生成一个 BigDecimal 并映射形成一个 Balance。唯一的新内容是在 validCheckingAccountGen❷ 中使用的 Arbitrary（任意）抽象 ❶，它使我们可以使用任意组合器生成一个 Balance。
- 生成账户编号：留意如何利用 choose 组合器生成特定范围内的整型值，以便可以使它成为一个有效的账户编号。账户编号的有效长度必须至少是 5。
- 生成帐户关闭日期：在 ❸ 中使用了一些 ScalaCheck 里有趣的数据生成功能。我们只能生成一个 None 的值，或者在生成传入的 seed 日期之后关闭日期（通常将开户日期作为 seed），但是可以控制生成数据的频率。在此处，我们指定要生成频率为 8 的有效关闭日期。ScalaCheck 将基于此分布生成数据。
- 生成有效的支票账户：这只是前面生成器的单子组合。在得到所有生成帐户所需的数据之后，使用智能构造器 checkingAccount 来构造一个有效的实例。智能构造器返回一个分离（\/），因此需要做适当的转换并对生成的帐户验证属性。在这一点上可能要重温一下清单 9.6 和 9.7，并确保已经掌握整个的帐户创建过程。

9.5.4 是否比基于 xUnit 的测试更好

正如在前面章节中看到的，属性迫使我们以模型代数行为的方式进行思考。在基于 xUnit 的测试中，需要使用一些手工数据的实例来测试场景。当测试复杂领域模型时，你认为哪一个更有用？以下列出了这些方法的优点和缺点。

- 直观：使用基于示例的手工数据来测试场景，不熟悉领域的人看起来更直观。我们将看到具体数据元素传递的函数或类，并对它们进行断言测试。这有助于我们更好地与底层实现相对应。基于 xUnit 测试具体用例在此先赢一局。
- 属性的思维：基于属性的测试迫使我们在模型的代数行为上更加努力地思考。

这通常会导致测试路径被更好地覆盖，因为大多数属性在本质上往往是横向切分并且映射到领域的业务规则上。这个练习也更贴近函数式编程的哲学，在这里我们也鼓励以代数方式进行思考。

- *领域规则存储*：丰富的属性作为领域规则的蓝图，这是一个无价的存储，它可以更好地验证领域模型。
- *数据生成*：在 xUnit 基于示例的方法中，可以手工编写数据，并用手动编写的少数数据来测试场景。而另一方面，通过基于属性的测试，可以生成海量的数据，并用整个数据集测试每个函数。我们可以控制这些数据的数量和分布，并在测试中获得更好的边缘情况和边界条件的覆盖。这可能是采用基于属性的测试所获得的最大优势。

9.6 总结

本章介绍了如何设计可测试的领域模型，然后使用基于属性的测试来验证模型的代数属性。以下是所涉及的主要结论：

- *可测试性与模块化密切相关*。需要确保在领域组件之间正确地分离关注点。
- *xUnit 及其限制*。还看到了基于 xUnit 测试的缺点，以及如何使用基于属性的技术作为替代。
- *基于属性的测试及其优点*。基于属性的测试允许通过生成器生成数据，所以我们对生成哪些数据有细粒度的控制。然后，就可以使用生成器来定义并验证领域模型的代数属性。

10 核心思想与原则

本章包括

- 本书涵盖的核心指导原则
- 领域建模趋势的概述

本章回顾了本书涵盖的一些基本思想、原则和术语。在前面的章节中，讨论了许多构建领域模型的技术，在个人和投资银行领域中，这些技术都有良好的响应性并且可以推导出纯函数。其中一些技术是核心原则，在进行建模时必须遵守它们。而其他一些更高级的技术会使代码更优雅、更高效并且模块化。我们在接下来的章节中会对它们进行回顾。

10.1 回顾

现在我们已经经历了如何以纯粹和组合结构的方式来思考领域模型的整个过程，让我们回顾一下构成整个实现思考过程核心的相关主题。每次开始实现一个领域模型时，这些内容都应该回顾一下。它们不仅仅是实现组件，它们中的许多成员已经形成了基本模式，在进行函数式领域建模时会成为我们思考过程的一部分。本

书已经反复提到，函数式方法类似于我们在数学中处理函数的方式——如果没有正确思考，就不会得到问题的理想解决方案。在许多情况下，都需要进行迭代。即使有很多年经验的成熟设计师在最终确定其预期的特定模型之前都依赖迭代和反馈。

我们需要记住的第一件事是领域模型属于问题领域。在开始实现解决方案模型之前，需要了解模型元素、领域词汇以及子系统边界。实现必须能够体现问题领域的所有行为，具有相似的清晰度并使用相同的通用业务语言（领域词汇表）。正如本书许多章节所提到的，该解决方案必须向用户呈现整洁、干净、领域友好并且没有任何底层实现的偶发复杂性的接口。我们讨论了通过使用各种函数式设计模式以及响应式原则来实现这些目标的技术。设计正确的抽象层次是其中的关键，在本章中将再次回顾这些模式和原则。我们可以随时修补实现的细节，但需要坚持核心原则。让我们来回顾一下其中比较重要的一些内容。

10.2 函数式领域建模的核心原则

在本节中，将回顾前几章的一些核心设计模式。讨论的重点是，当使用函数式编程进行领域建模时，为什么要遵守这些原则。本章不再大谈函数式编程的好处，我相信大家现在已经很熟悉了。但是为了在围绕纯函数设计领域模型时获得最大的力量，我们需要遵守一些基本原则。

10.2.1 表达式思想

一方面，表达式会有值；另一方面，声明会产生副作用。一个表达式生成的值可以在函数中传递，生成更大的表达式。从较小的表达式生成较大的表达式，逐步演化领域行为，以便可以更好地与其他行为进行组合。让我们回顾一下 4.2.3 节中讨论的例子：

```
object App extends AccountService {
  def op(no: String) = for {
    _ <- credit(no, BigDecimal(100))
    _ <- credit(no, BigDecimal(300))
    _ <- debit(no, BigDecimal(160))
    b <- balance(no)
  } yield b
}
```

函数 op 通过计算 for 表达式来生成一个值。for 表达式中的每个语句都是一个生成单独值的表达式，并通过一系列 flatMap 和 map 将它们链接在一起生成最终的值。表达式思想可以帮助我们在组合方面展开思考，使领域行为的进化更加自然。使用组合器来抽象作用，就可以在不打破表达式链条的情况下组合更复杂的行为，

而不是过早地承担副作用。[1]

10.2.2 早抽象，晚赋值

在3.2.1节中，已经看到了如何区分计算和值。计算是对值的抽象，例如，Try对失败进行了抽象。在实现领域模型时，不要过早提交值，否则会失去与其他计算组合特定行为的能力。如果有一个debit函数，它在面对账户余额不足时会产生失败，并返回一个Try或者scalaz.\/作为一个计算，而这个计算抽象了操作的结果。或者，也可以返回一个表示成功或失败的值。但在这样做时，就不能将debit操作与下游其他操作串联起来。例如，不能像前一节那样把它们组合成一个for表达式。所以，当到达计算序列的结尾时，再提交值。

10.2.3 使用合适的抽象

乍一看，这似乎有点不合常理。但是考虑到抽象越强大，它就越专业，所以我们可以在不太普遍的情况下使用它。如果使用的抽象比所需要的要强大得多，就会失去可重用的部分。在4.2.3节讨论单子和applicative作用的差异时已经对此展开了很多的讨论。Monad比applicative更强大，但可重用性较差。最终得出的结论是，只要可以就应该使用applicative，这样抽象就可以在更多上下文中得到重用。在函数式编程的上下文中已经提到过这个原则，但这在任何建模或设计中，都是一个应该遵循而且非常有用的原则。

10.2.4 发布要做什么，在组合器中隐藏如何做

函数式编程的核心优势之一就是提供了合适的工具来声明我们想要做什么，而不是如何做。*how*的部分可以被抽象在函数中，而这些函数又可以与实现*what*部分组合在一起。考虑一下4.4.3节清单4.10中的例子。tradeGeneration的实现看起来像是来自业务规范文档的复件，它提到了需要遵循哪些步骤来生成交易。执行的快乐路径和异常路径中组合在一起的各种步骤，如何被抽象在组合器andThen内部。如果看完了4.4.3节中的例子，就一定会注意到andThen是在Kleisli类中定义的高阶函数，并使用一个monad的代数在API中提供所期望的作用。所有这些都来自于组合函数和重用已有代数的能力，这也有助于使实现简洁且富有表现力。[2]

1 就如同在3.2节中用Try或分离来抽象失败一样。

2 andThen重用了monad的代数。

```
def tradeGeneration(
  market: Market,
  broker: Account,
  clientAccounts: List[Account]
) = {
  clientOrders andThen
    execute(market, broker) andThen
      allocate(clientAccounts)
}
```

10.2.5 从实现中解耦代数

这是系统设计最基本的原则之一。接口会被发布到客户端，因此不能随意更改。但是底层实现可以改变，因此就需要与已发布的接口解耦。本书的重点在于基于代数的设计，我们所说的代数就是发布给客户端的 API 接口。对于一个代数，可以有多个实现，但是需要很小心地提供两者之间正确的隔离级别。第 5 章中为实现这一目标的提供了许多方法：

- 使用基于特征（trait）的模块化组合，在对象的具体实现中结束（5.2 节）。
- 类型类模式（5.3 节）。
- free monad 和解释程序模式（5.5 节）。

模块化是使领域模型在面对变化时依然可组合和可管理的主要因素之一。可以根据每个要求选择使用这些模式。

10.2.6 隔离边界上下文

模块自然会导致边界上下文。边界上下文是具有自己业务通用语言、实体和类型的独立领域模型。通常来说，一个领域模型的内部都有多个模型，每个模型都建模了一个单独的边界上下文。识别边界上下文可能是在做领域建模时应该做的第一个迭代练习。我们在第 5 章中详细讨论了边界上下文，也看到了它们之间进行沟通的各种方式。在许多情况下，我们会意识到单独的边界上下文通常可以演化为微服务——它作为整个模型的单元，具有单独的数据类型语义、实体、关系以及领域词汇表。

10.2.7 偏向 future 而不是 actor

在当今的社区中，actor 比 future 更有市场。但必须记住，future 可以组合从而为建立更大的行为打下更好的基础。在第 6 章已经详细讨论了响应式系统的设计原理。当寻求构建异步抽象的方案时，总是会先想到 future。[1] 这并不意味着 actor 在领

1 详细内容参见第6章中所讨论的scalaz.concurrent.Task。

域模型中没有任何作用。第 6 章提供了一些值得使用 actor 的应用场景。毕竟，当开始实现复杂领域模型的工作时，手上有更多的工具总是件好事。

10.3 展望未来

本书重点强调了函数的强大以及它们在泛型抽象中的组合性，这构成了领域模型的核心基础。我们需要思考模型的行为，而不是专注于对象。原因很简单，行为（函数）可以用数学的方式进行组合。将模型元素映射到数学世界的那一刻起，就开辟了数学验证模型的世界。Scala 不是那种纯粹的语言，Haskell 这样的语言更纯粹，Haskell 中的函数很像数学中的方程式，可以对功能做等式推导并证明它们的行为方式是符合我们期望的。Haskell 使等式推导比 Scala 更可行的主要原因是 Haskell 类型系统支持更严格的副作用与纯逻辑之间的分离。Haskell 中的类型系统将确保当作数学等式操作的函数是真正符合代数的，而且不会在其他非预期的副作用中溜达。如果在编程风格中采用某些规定，就会更接近 Scala。但是使用 Haskell，推导部分会比 Scala 更好用一些。用 Haskell 这样的语言建模领域绝对是期待的未来趋势之一。

另一个最近正在被开发的内容是依赖类型，它可以在开发静态验证的领域模型中添加很多值。许多使用过程逻辑编码的领域约束或规则可以被编码为类型系统本身的一部分。Idris（www.idris-lang.org）是一个提供这些功能的依赖类型语言。Haskell（https://wiki.haskell.org/Dependent_type）也开始开发依赖类型的功能。在 Scala 中，使用 Shapeless（https://github.com/milessabin/shapeless）做了许多这样的事情，尽管方式上显得略微冗长。更强大的类型系统意味着更简洁的领域逻辑编码，这又导致需要编写的测试代码越来越少。

随着模型变得越来越复杂，作为建模者，我们面临着使这些模型更有响应性的压力。我们已经看到响应式一词如今变得越来越主流。我们有响应式宣言，也看到了更多的库和框架声称它们是响应式的。在第 6 章和第 7 章中，介绍了很多关于如何借助 Scala 提供的各种并发并行工具使模型具有响应性的内容。我们还专门讨论了 Akka（http://akka.io），它已经成为在 JVM 上开发响应式应用最受欢迎的工具包之一。

响应式应用开发的一个趋势是使用流来传递领域服务。我们已经介绍了 Akka Streams，它是来自 Akka 的一个备受欢迎的流媒体库。大家可能会想为什么流在领域建模的上下文中变得如此受欢迎。原因就在于我们会遇到越来越复杂的行为，这些行为涉及大量数据流作为核心业务的组成部分，然后以非阻塞的方式应对领域服务越来越高的响应性要求。就在前几天，我们在一个独立的分析平台上做了 map-reduce 的批处理。提供实时解决方案的需求正在迅猛增长，而流处理是达成该目标的主要技术之一。Apache Spark（http://spark.apache.org）、Apache Flink（http://flink.

apache.org）、Apache Samza（http://samza.apache.org） 和 Apache Storm（http://storm.apache.org）已经成为这个联盟里的主要玩家，而且参与者还在日益增加。领域模型需要处理流并且与这些库集成。讽刺的是，挑战恰恰来自要遵守软件工程的古老话题——关注点分离。我们需要考虑如何保持核心领域行为与技术相关点（如流或批处理）的分离。我们会再次发现，函数式编程提供的模块化原则将会帮助到我们。用 Scala 作为实现语言的 Streaming 平台（如 Spark 或 Flink）都采用了函数式编程原则。这不是没有理由的，最终我们会发现函数式编程的数学基础为领域模型的模块化提供了最强的基础。

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

传统的分布式应用不会切入微服务、快速数据以及传感器网络的响应式世界。为了捕获这些应用的动态联系以及依赖，我们需要另外一种不同的方式对系统进行领域建模。由纯函数构成的领域模型是以一种更加自然的方式来反映一个响应式系统内的处理流程，同时它也直接映射到相应的技术和模式，比如 Akka、CQRS 以及事件溯源。

本书首先介绍了函数式编程和响应式架构的相关概念，然后逐步地在领域建模中引入这些新的方法。同时本书提供了大量的案例，在项目中应用这些概念时，可作为参考。

本书包含的内容：

- 现实的库和框架
- 建立有意义的可靠性保证
- 将领域逻辑与副作用隔离
- 响应式设计模式的介绍

读者可以很快适应函数式编程以及传统领域建模。所有案例都使用 Scala 语言编写。

Debasish Ghosh，软件架构师，是使用 Scala 和 Akka 进行响应式设计的先行者。他同时也是 *DSLs in Action* 一书的作者，该书由 Manning 出版社于 2010 年出版。

将 3 种不同的工具——领域驱动设计、函数式编程以及响应式原则——整合到一起的实践之路。

——Jonas Bonér
Akka 创始人

现实领域驱动设计的新式方法。

——Barry Alexander
Gap 自动化工程师

作者用清晰的讲述方式证明了他在 DDD 实践上的杰出才能。

——Cosimo Attanasi
ER Sistemi 软件开发人员

提供了一个学习 DDD 和 FP 的独特视角。

——Rintcius Blok
Cake Solutions 软件工程师

现代软件设计技术的经典之作。

——William Wheeler
Java/Scala/Akka 开发人员

博文视点Broadview
新浪微博 weibo.com
@博文视点Broadview

MANNING

责任编辑：张春雨
封面设计：吴海燕

上架建议：软件架构/软件工程

ISBN 978-7-121-32392-8

定价：79.00元